# TotalView

## User's Guide

Version 4.1
June 2000

ETNUS

# Contents

## 9 Visualizing Data



## 10 Troubleshooting



## 11 X Resources



## 12 TotalView Command Syntax



## 13 TotalView Debugger Server Command Syntax



## A Compilers and Environments

# About This Book

This guide describes how to use TotalView®, a source-level and machine-level debugger with an easy-to-use interface and support for debugging multiprocess programs. The guide assumes that you are familiar with programming languages, the UNIX operating systems, the X Window System, and the processor architecture of the platform on which you're running TotalView.

This guide covers using TotalView on any platform. Most of the examples and illustrations in this guide show TotalView running on a workstation. To learn about the specifics of running TotalView on your platform, refer to Appendix A, "Compilers and Environments," on page 319, Appendix B, "Operating Systems," on page 329, and Appendix C, "Architectures," on page 343.

## Supported Platforms

TotalView is available for a variety of platforms and can be used to debug programs on the native platform or on remote systems, such as parallel processors, supercomputers, or digital signal processor boards.

If TotalView is not yet available for your system configuration, please contact Etnus® about porting TotalView to suit your needs:

**Etnus Inc.**
111 Speen Street
Framingham, MA 01701-2090
Internet E-mail: **info@etnus.com**
1-800-856-3766 in the United States
(+1) 508-875-3030 worldwide

## Reporting Problems

Please contact us if you have problems installing TotalView, questions that are not answered in the product documentation or on our Web site, or suggestions for new features or improvements.

Internet E-Mail addresses: **support@etnus.com**
United States Phone Number: 1-800-856-3766
Worldwide Phone Number: (+1) 508-875-3030

If you are reporting a problem, please include the following information:

- The **version** of TotalView
- The **platform** on which you're running TotalView
- An **example** that illustrates the problem
- A **record** of the sequence of events that led to the problem

See the TOTALVIEW RELEASE NOTES for complete instructions on how to report problems.

# Conventions

The following table describes the conventions used in this book:

Table I: Book Conventions

| Convention | Meaning |
|---|---|
| [ ] | Brackets are used when describing parts of a command that are optional. |
| *arguments* | Within a command description, text in italic represent information you type. Elsewhere, italic is used for emphasis. You will not have any problems distinguishing between the uses. |
| **Dark text** | Within a command description, **dark text** represent key words or options that you must type exactly as displayed. Elsewhere, it represents words that are used in a programmatic way rather than their normal way. |

# Chapter 1

# TotalView Features

The TotalView® debugger is part of a suite of software development tools for debugging, analyzing, and tuning the performance of programs, including multiprocess multithreaded programs.

This chapter highlights:

- TotalView's advantages
- TotalView's windows
- Examining source and machine code
- Controlling processes and threads
- Using action points
- Examining and manipulating data
- Visualizing array data
- Distributed debugging
- Debugging multiprocess and multithreaded programs
- Context-sensitive help

## TotalView Advantages

TotalView provides many advantages over conventional UNIX debuggers such as **dbx**, **gdb**, and **adb**:

- You can learn TotalView quickly and be more productive because of its Graphical User Interface (GUI) that is based on the X Window System. TotalView provides windows, pop-up menus, and context-sensitive help.
- TotalView's interface lets you see a lot of useful information without entering commands.

- You can debug *multiprocess multithreaded* programs. TotalView displays each process in its own window, showing the source code, stack trace, and stack frame for one or more threads in the process.
- You can display all process windows simultaneously and perform debugging tasks across processes.
- TotalView's distributed architecture lets you debug *remote* programs over the network.

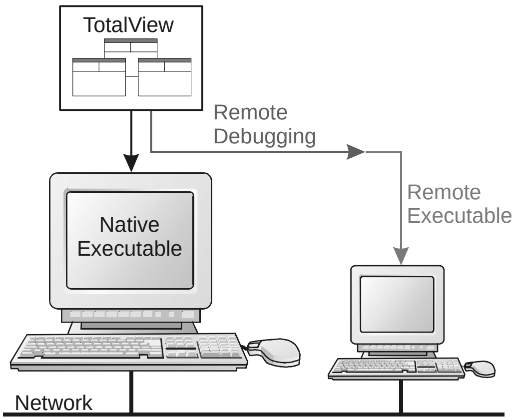



FIGURE 1: **Debugging a Remote Program with TotalView**

TotalView can manage multiple remote programs and multiprocess multithreaded programs simultaneously, as shown in Figure 2.

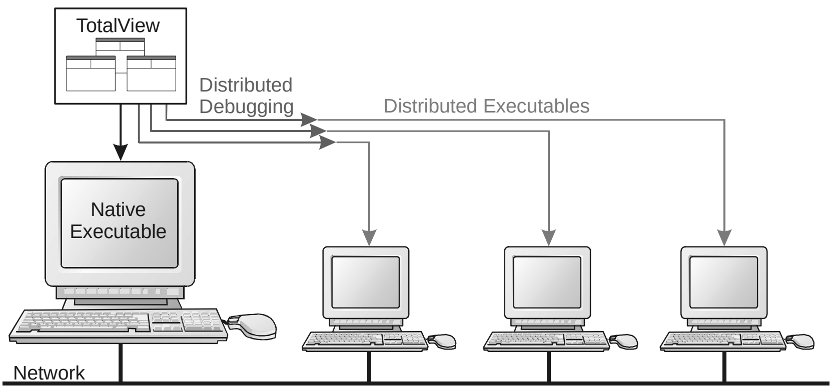



**Figure 2: Debugging a Distributed Program with TotalView**

- Parallel and distributed programs run in many processes, and your debugger must know about them. When you start TotalView as part of an HPF, MPI, PE, or PVM, application, TotalView automatically detects and attaches to these processes. This is called *automatic process acquisition*.
- Because TotalView lets you attach to running processes, you can debug processes that were not started under TotalView's control.
- TotalView lets you temporarily add source code statements to the program you are debugging. On some platforms, you can even add machine code statements. This feature saves time when you are testing bug fixes.
- If the code you are debugging was not compiled using the **–g** option or if you do not have access to the program's source file, TotalView lets you debug its machine-level code.
- TotalView's Command Line Interface lets you enter commands directly in an xterm window when you find yourself unable to use the GUI. (The CLI is described in the CLI USER'S GUIDE.)

## TotalView Windows

TotalView displays extensive information in its windows, as shown in Figure 3 on page 4. This figure shows four windows:

| | |
|---|---|
| Root | Lists the name, location (if a remote process), process ID, status, and, optionally, a list of threads for each process being debugged. It also shows the thread ID, status, and current routine executing for each thread. |
| Process | Displays information about a process and a thread within that process. It also shows the stack trace, stack frame, and source code for the selected thread in a series of separate panes. Optionally, it displays disassembled machine code or interleaved source code and disassembled machine code. |
| Process Groups | Shows the process groups for all multiprocess programs you are debugging. |
| Variable | Contains the address, data type, and value of a local variable, register, or global variable. It also shows the values (and optionally, the machine-level instructions) stored in a block of memory. |

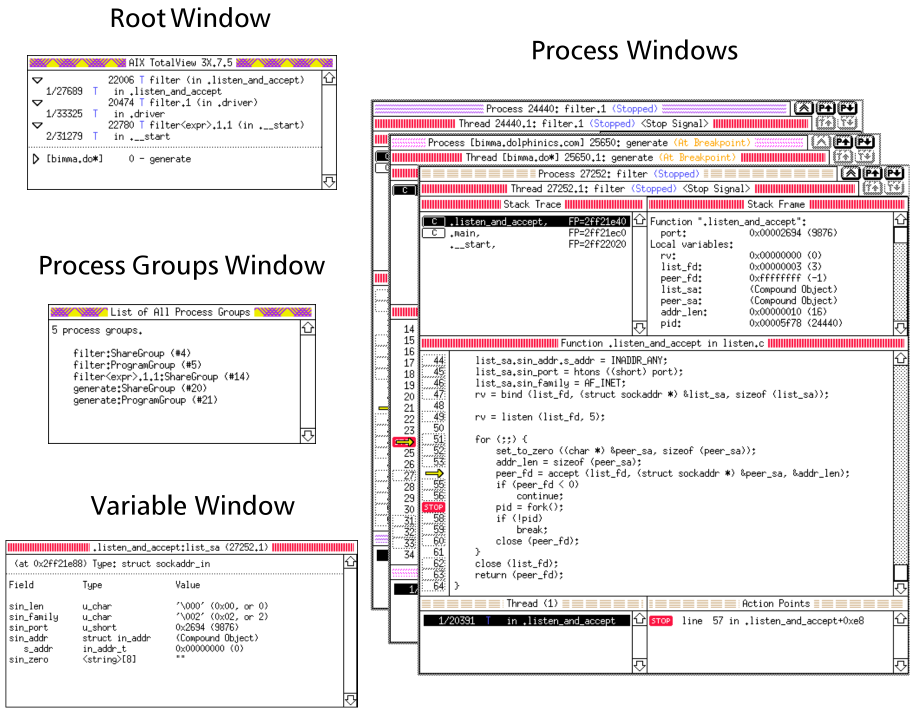



**Figure 3: Sample TotalView Session**

# Multiprocess Programs

TotalView has special features for debugging multiprocess programs.

- **Process groups**

  TotalView treats multiprocess programs as process groups. When debugging multiprocess programs, you can view information about all process groups and can view information about a multiprocess program. Using TotalView, you can start and stop individual process groups.

- **Separate windows for each process**

  Each process has its own process window displaying information for that process. You can monitor the status, thread list, breakpoint list, and source code for *each* process. You do not have to display all the process windows in a multiprocess program; instead, you can choose which process windows to display.

- **Sharing of breakpoints among processes**

  You can control if a breakpoint is shared among child processes and if all processes in the group stop when any process in the group reaches a breakpoint.
- **Process barrier breakpoints**

  In addition to "normal" breakpoints, TotalView allows you to create process barrier breakpoints. A process barrier breakpoint differs from a regular breakpoint in that it holds every process that reaches the barrier until all processes in the group reach it. When the last process in the group reaches the barrier, TotalView releases all of these held processes. Because TotalView will not let you release a held process, a barrier lets you synchronize a group of processes to the same location.
- **Process group-level single-stepping**

  TotalView allows you to single-step groups of processes using one command.
- **Single event log containing information for all processes**

  TotalView logs significant events about each process being debugged. Thus, you can view the history of your entire debugging session by scrolling through the Event Log Window.
- **Automatically attach to child processes**

  If a program calls **fork()** or **execve(),** TotalView automatically attaches to the child process and includes it in the process group.
- **Multiple symbol tables**

  If you are debugging more than one executable at a time, TotalView automatically handles the symbol table for each.

## Multithreaded Programs

While the way in which operating systems implement threads vary, most share the following characteristics:

- **Shared address space**

  The threads share an address space (memory) with other threads. They can read and write the same variables and can execute the same code.
- **Private execution context**

  Each thread has its own general-purpose and floating-point registers.

- **Thread private data**

  Some operating systems allow a program to declare thread private data. This information provides each thread with its own copy of the variable. Changes made by one thread to its private variables are not seen by other threads.

- **Private execution stack**

  Each thread has an address space reserved for its execution stack. However, one thread's stack can be read and written by other threads sharing the address space.

TotalView supports debugging threaded applications on a variety of operating systems. On some of these systems, a process consists of an address space and a list of one or more threads. Other operating systems implement tasks or threads running in the computer's memory space and do not support multiple processes or address spaces on a single machine.

Because the ways operating systems handle threads differ, TotalView implements a general model of address spaces and execution contexts. A TotalView *thread* refers to thread or task with an execution context, and *process* refers to an address space or computer memory that can run one or more threads.

## Controlling Processes and Threads

TotalView offers a full range of methods for controlling processes and threads. Using TotalView, you can:

- **Start and stop processes and threads**

  You can start, stop, resume, delete, and restart your program.

- **Attach to existing processes**

  TotalView lets you examine processes that are not running under its control. Attaching to one of these processes is as easy as diving on it.

- **Examine core files**

  You can load a core file and examine it in the same way as any other executable. Or, you can load a core file anytime during while debugging.

- **Reload the executable file**

  After editing and recompiling a program, you can reload it.

- **Single-step your program**

  You can single step through your program or step over function calls. You can tell your program to execute to a selected source line or instruction or continue executing until a function completes its execution. TotalView supports process level, process group level, and, on some systems, thread level single stepping.

- **Change the way TotalView handles signals**

  TotalView lets you tailor how signals are handled. For example, it can stop the process and place it in a stopped or error state, sending the signal on to the process, or discarding the signal.

- **Change the program counter (PC)**

  You can change the value of the PC to resume execution at a different point in the program.

## Using Action Points

TotalView provides a broad range of action points. (Action points are places in a program where you stop execution or evaluate an expression.)

**Action points:** You can set, delete, suppress, unsuppress, enable, and disable action points at the source and machine levels. TotalView lets you set the following action points:

- **Breakpoints**: stop execution when a statement or instruction executes.
- **Barrier breakpoints**: hold other threads until all threads in a group reach a "barrier" statement or instruction.
- **Conditional breakpoints**: only perform an action if a code fragment (expression) is satisfied.
- **Evaluation points**: execute code you create at a statement or instruction.
- **Watchpoints**: monitor when changes occur to a variable's value.

**Expressions and code fragments:** TotalView lets you write and evaluate fragments of code, including function calls used by the current process. While differences exist between platforms, you can write fragments in C, C++, Fortran, and assembler. On most platforms, TotalView compiles code fragments.

# Examining and Manipulating Data

TotalView provides many ways for you to examine your code. Here are two methods.

- **Search for functions**

  You can search for functions using a dialog box.

- **Dive on function**

  You can click your right mouse button on a a function's name to tell the debugger to display the function's source code in the Source Code Pane of the Process Window. (This process is called *diving*.)

Similarly, TotalView lets you examine and manipulate data in your program, as follows:

- **Dive on variables**

  You can dive into a variable in the same way that you dive into a function. That is, you click the right mouse button while the cursor is over the variable. (You can also dive into a variable using a menu command.) TotalView lets you examine local variables, registers, global variables, machine-level instructions, and areas of memory. In all cases, TotalView displays this information in a separate variable window.

- **Change types**

  You can alter a variable's type to display the data in different formats.

- **Change values**

  You can edit the value of a variable or a memory location, changing it for the current running process.

- **Laminate variables**

  You can examine the value of a variable across multiple processes and multiple threads in a single data window. (This ability to display the multiple values of a variable is called *lamination*.)

- **Examine array data**

  You can filter array data to look for elements that match a filter expression. You can also sort data and tell TotalView to display statistical information about an array's contents.

## Visualizing Array Data

The TotalView Visualizer allows you to graphically view array data in the programs you are debugging. This gives you an overall picture of your data and helps you to find incorrect data quickly and easily.

**NOTE** The Visualizer is not available on all platforms.

You can visualize array data using the:

- **Visualize variable window menu command**

  This command tells TotalView and the Visualizer to show you a visual snapshot of the array data listed in the window. Each time you visualize the same array, the Visualizer image is updated.
- **$visualize statement**

  You can use the **$visualize** statement from the Expression Evaluation Window and within evaluation action points to visualize one or more data sets within a single expression. Each time TotalView evaluates an expression, the Visualizer updates the images that it is displaying. This allows you to animate the visual representation of your data.

TotalView also allows you to use your own visualization program.

## Distributed Debugging

TotalView provides a distributed architecture that supports many different operating environments, including:

- Remote programs running on a separate machine from TotalView.
- Distributed programs running on a set of homogeneous machines.

  NOTE **Distributed debugging requires that all machines have the same architecture and operating system.**
- Multiprocess programs running on a multiprocessor machine.
- Multiprocess programs running on a cluster of homogeneous machines.
- Client-server programs with the server running on one machine type and the clients running on another machine type.

The machine on which TotalView is running is known as the *host machine*, while the machine on which the process being debugged is running is the *target machine*. The host and target machines can be the same machine.

If the host and target machines are different, TotalView starts a process on each remote target machine. TotalView communicates with this process using standard TCP/IP protocols.

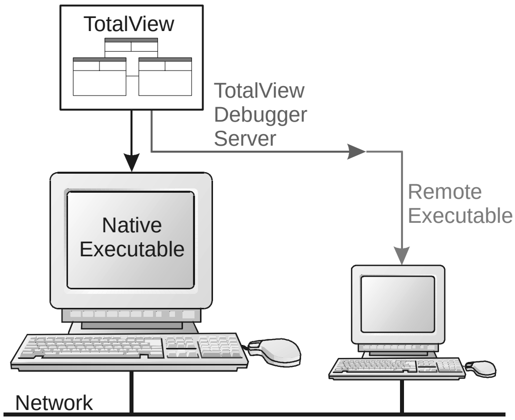



**Figure 4: TotalView Debugger Server**

Debugging distributed programs does not differ from debugging non-distributed programs: TotalView offers the same set of rich features to both.

Depending on the platform, TotalView can debug programs that use the HPF, MPI, IBM Parallel Environment (PE), OpenMP, pthreads, and Parallel Virtual Machine (PVM) libraries.

## Context-Sensitive Help

You can request help from any window being displayed. The **Help** command displays context-sensitive information about the current window or dialog box or the debugging operation you are currently using. TotalView displays the information in a separate help window.

Chapter 2

# TotalView Basics

This chapter introduces you to the TotalView interface and describes how you:

- Compile your program
- Start TotalView
- Use the mouse buttons and menus
- Get online help
- Use the windows
- Dive into objects
- Edit text
- Search for text strings
- Use the spelling corrector
- Save a window's contents
- Exit from TotalView

## Compiling Programs

Before you start TotalView, compile your source code with the **–g** compiler option, which generates symbol table debugging information. For example:

`cc –g` *source_program* `–o` *executable*

For more information on compiling your program for TotalView, see "*Compiling Programs*" on page 29.

On some platforms, you may need to use additional compiler options. Refer to Appendix A, "*Compilers and Environments*" on page 319 for more information.

TotalView also lets you debug programs that were not compiled with the **–g** option or programs for which you do not have source code. For more information, refer to "*Examining Source and Assembler Code*" on page 118.

When TotalView reads a file, it uses the file's extension to determine the programming language that you used to write the file's contents, as shown in the following table.

TABLE 1: Source Language Mapping

| File Extension | Source Language |
|---|---|
| **.cxx**, **.cc**, **.cpp**, **.C**, **.hxx**, **.H** | C++ |
| **.F**, **.f**, **.F90**, **.f90** | FORTRAN 77 or Fortran 90 |
| **.hpf**, **.HPF** | HPF |
| All others | C |

TotalView identifies a program as FORTRAN 77 or Fortran 90 when:

- The compiler explicitly specifies the language in the debug information.
- The source filename has an **.f90** or **.F90** suffix.
- The code uses Fortran 90 features such as assumed shape arrays or pointers.

If TotalView cannot identify a source file's language, it assumes that the source language is C. If this is a problem, you'll need to change the file's extension to one that TotalView recognizes.

## Starting TotalView

Depending on the kind of program you are debugging, there are several way to start TotalView. The simplest method uses the **totalview** command and your program's name:

**totalview** *exectuable*

A similar command can be used to start the CLI:

**totalviewcli** *executable*

The CLI is described in the CLI USER'S GUIDE.

The program you are debugging may require options or that you invoke TotalView in a different way. For more information, see:

- "*Starting the TotalView Debugger*" on page 30.
- Chapter 5, "*Setting Up Parallel Debugging Sessions*" on page 69.
- The **totalview** command syntax, described in Chapter 12, "*TotalView Command Syntax*" on page 299.

# Using the Mouse Buttons

TotalView uses the buttons on your three-button mouse as follows:

TABLE 2: **Mouse Button Functions**

| Default Position | Button | Purpose | How to Use It |
|---|---|---|---|
| Left | Select | Select or edit object, scroll in windows and panes | Move the pointer over the object and click the button |
| Right | Dive | Dive into object to display information about it | Move the pointer over the object and click the button |
| Middle | Menu | Display pop-up menu | Move the pointer into the window and hold down the button |
| | | Select command from menu | Move pointer down the menu until the desired command is highlighted, and release the button |
| | | Leave menu without selecting command | Move the pointer off the menu and release the button |

In the tag field area (the area on the left containing source code numbers) of the Source Code Pane, the Select (left) button has a special function: selecting the line number sets a breakpoint at that line. TotalView responds by displaying a **STOP** icon in the tag field.

Selecting the **STOP** icon removes the breakpoint. If an evaluation or event point was set (indicated by an **EVAL** or **ELOG** icon), selecting the icon disables it. For more information on breakpoints, evaluation points, and event points, refer to Chapter 8, "*Setting Action Points*" on page 195.

## Using Menu and Keyboard Commands

Each window has its own set of commands that are invoked using a pop-up menu. These commands let you examine and manipulate the displayed information. Figure 5 shows an example of the Process Window menu and one of its menu. To display a pop-up menu in the current window, click the middle mouse button.

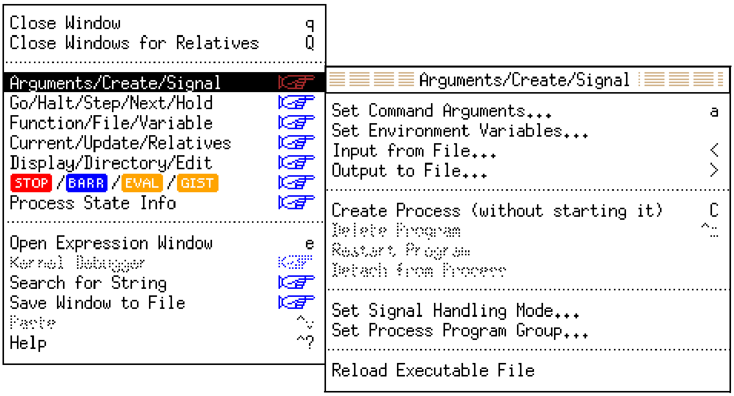



**Figure 5: Pop-up Menu and submenu**

Many commands have keyboard shortcuts. For example, typing the letter **q** while in the Process Window invokes the **Close Window** command. Keyboard shortcuts are shown to the right of the menu command.

On the far right size of many menus is a hand with a finger pointing to the right. Placing your cursor on these lines and dragging your mouse to the right tells TotalView to display a menu with additional commands. Note that if a menu item is dimmed, the item is currently disabled.

The following commands are only available from the keyboard:

| | |
|---|---|
| Ctrl-C | Cancels a single-step operation and other time-consuming operations, such as searching for a string. |

| | |
|---|---|
| **Ctrl-L** | Refreshes the current window. |
| **Ctrl-Q** | Exits from TotalView. |
| **Ctrl-R** | Raises the Root Window. |
| **Shift-Return** | Exits from the field editor that you are using to edit text. |

# Getting Help

You can request help from any TotalView window or dialog box by selecting the **Help** command from the pop-up menu or by pressing **Ctrl-?**. When you request help, a separate Help Window appears. To close the Help Window, select the **Close Window** menu command.

# Using the Primary Windows

After starting TotalView with the name of the program being debugged, two windows appear:

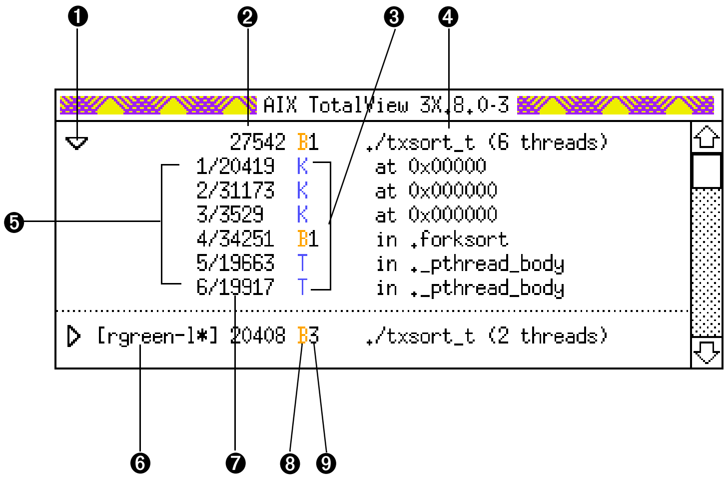
- The *Root Window* displays a list of all the processes that you are debugging, and optionally a list of threads for each process. Initially, the Root Window only contains the name of the program being debugged.
- The *Process Window* displays the thread list, action point list, and the selected thread of the process you are debugging. It also displays the source code, stack frame, and stack trace of the selected thread in that process. Initially, this window only contains your program's source code. Other information is added as your program executes.

Figure 6 and Figure 7 show the Root and Process Windows.

## Starting a Process

To start a process:

1. Move your cursor to the Process Window.
2. Set a breakpoint in the source code by selecting a boxed line number.
3. Type the keyboard accelerator **g** (for the **Go Process** command). The process starts running and then stops at the first breakpoint set.




❶ Collapse/expand toggle
❷ Process ID (pid)
❸ Thread status
❹ Program name
❺ Thread list
❻ Remote process location
❼ Thread ID (tid/systid)
❽ Process status
❾ Thread Status

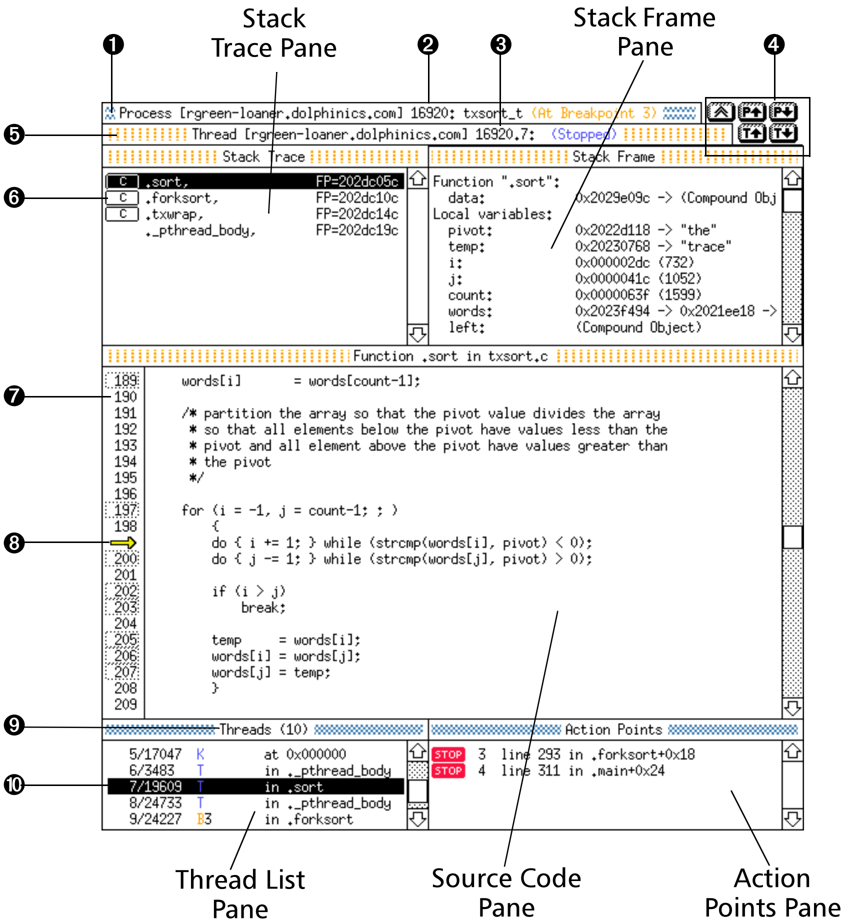
**Figure 6: Root Window**

When debugging a remote process, TotalView displays an abbreviated version of the hostname on which the process is running within square brackets in the Root Window. The full hostname appears in square brackets in the title bar of the Process Window. In Figure 7, the process is running on the machine **rgreen-loaner.dolphinics.com**, which is abbreviated to **[rgreen-l*]** in the Root Window.

As you examine the Process Window in Figure 7, notice the following:

- The thread ID shown in the Root Window and in the process's Thread List Pane is the TotalView assigned logical thread ID (or *tid*) and system assigned thread ID (or *systid*). On systems such as Compaq Tru64 UNIX where the *tid* and *systid* values are the same, TotalView displays only the *tid* value.

  In other windows, TotalView uses the value *pid.tid* to identify a process's threads.




❶ Process status
❷ Process ID (pid)
❸ Process and thread ID (pid.tid)
❹ Navigation controls
❺ Thread status
❻ Language of routine
❼ Tag field area
❽ Current PC
❾ Thread count
❿ Selected thread

**Figure 7: Process Window**

- The *Thread List Pane* shows the list of threads that currently exist in the process. The number in the Thread List Pane title (❾) is the number of threads that currently exist in the process. When you select a different thread in this list, TotalView updates the Stack Trace Pane, Stack Frame

Pane, and Source Code Pane to show the information for that thread. When you dive on a different thread in the thread list, TotalView finds or opens a new window displaying information for that thread.

Holding down the Shift key when you dive tells TotalView to open a new Process Window focused on that thread.

- The *Stack Trace Pane* shows the call stack of routines that the selected thread is executing. You can move up and down the call stack by selecting the routine (stack frame). When you select a different stack frame, TotalView updates the Stack Frame and Source Code Panes to show the information about the selected routine.
- The *Stack Frame Pane* displays all the function parameters, local variables, and registers for the selected stack frame.
- The information displayed in the Stack Trace and Stack Frame Panes reflects the state of the process when it was last stopped. Consequently, this information is not up-to-date while the thread is running.
- The left margin of the *Source Code Pane*—called the tag field area—displays line numbers. You can place a breakpoint at any source code line that generated object code. (These places are indicated by a boxed line number.) The arrow in the tag field indicates the current location of the program counter (PC) within the selected stack frame. See Figure 8.


**Figure 8: Process Window Navigation Controls**

- In multiprocess and multithreaded programs, each thread has its own point of execution. This means that each thread's Process Window has its own unique program counter (PC). Therefore, when you stop a multiprocess or multithreaded program, the routine selected in the Stack Trace Pane for a thread depends on the thread's PC. When you stop the program, some threads can be executing in one routine, while others might be executing elsewhere.
- The *Action Points List Pane* shows the list of breakpoints, evaluation points, and watchpoints for the process.
- The navigation control buttons in the upper right-hand corner of the Process Window allow you to easily navigate through the processes and threads being debugged.

## Sizing Process Window Panes

You can change the size of the panes in the Process Window. If you do not want to see a pane, you can size the pane to a zero size. Here is how you resize a pane:

1. Move the mouse cursor over the edge of the window pane until the cursor with crossed arrows appears.

**Figure 9: The Sizing Cursor**

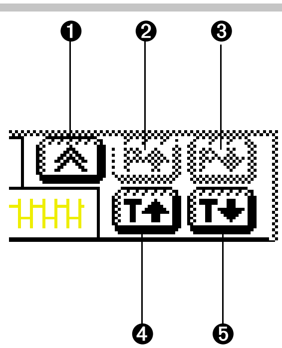
2. Hold the left mouse button down and drag the edge until the pane is the size you want it to be.

## Navigating in the Process Window

The navigation control buttons, located in the upper right corner of the Process Window, allow you to easily navigate through processes and threads. Using these buttons you can

- Move up and down the list of processes you are debugging.
- Move up and down the list of threads in a process.
- Go back to the previous contents of the Process Window.

Figure 10 shows the navigation controls available in the Process Window.




❶ Go back
❷ Previous process
❸ Next process
❹ Previous thread
❺ Next thread

**Figure 10: Process Window Navigation Controls**

## Navigating in the Root Window

You can also navigate through processes and threads from the Root Window. In general, selecting a process or thread with the left mouse button does not open a new window. However, diving on a process or thread with the right mouse button opens a new Process Window if an exactly matching process/thread combination is not found. Finally, holding down the Shift key when you dive always opens a new window.

**NOTE** **Whenever a process or thread is replaced in the Process Window, the previous contents of the window are *pushed* onto a stack. The Go Back button *pops* the stack so that TotalView displays the previous contents of the Process Window.**

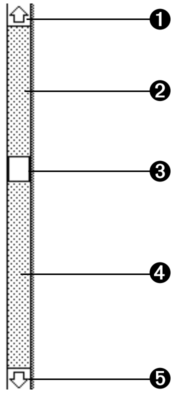
Here is a summary of how you select and dive on threads and processes:

- When you *select a process in the* Root *Window*, TotalView finds or opens a Process Window for that process. If it cannot find a matching window, TotalView replaces the contents of an existing Process Window and shows you the selected process.
- When you *dive on a process in the* Root *Window*, TotalView finds or opens a Process Window for that process. Holding down the Shift key when you dive tells TotalView to open a new Process Window focused on that process.
- When you *select a thread in the* Root *Window*, TotalView finds or opens a Process Window for that process and show you the selected thread. If a matching window cannot be found, it will replace the contents of an existing Process Window and show you the selected thread.
- When you *dive on a thread in the* Root *Window*, TotalView finds or opens a Process Window for that process and thread combination. Holding down the Shift key when you dive will force TotalView to open a new Process Window focused on that thread.

# Scrolling Windows and Fields

## Scrolling Windows

You can use the scroll bars to scroll through the information in TotalView windows and panes, as shown in Figure 11.




❶ Up arrow
❷ Page-up region
❸ Elevator box
❹ Page-down region
❺ Down arrow

**Figure 11: Scroll Bar**

- To scroll one line at a time, click the Select (left) mouse button on the up or down arrows (at the top and bottom of the scroll bar).
- To scroll one page at a time, click the Select mouse button above or below the elevator box inside the scroll bar.
- To scroll an arbitrary amount, hold down the Select mouse button and drag the elevator box inside the scroll bar.

To scroll continuously by line or by page, you can hold down the Select mouse button instead of clicking it. If TotalView scrolls too fast or too slow, you can adjust the scrolling speed using X resources. Refer to "TOTALVIEW*SCROLLLINESPEED" on page 289 for further information.

You can also scroll windows using the keys on your keyboard's numeric keypad, as follows:

| | |
|---|---|
| ↑ | Scrolls up one line. |
| ↓ | Scrolls down one line. |
| **Meta-↑** | Scrolls up one page. |
| **Page up** | Scrolls up one page. |
| **Meta-↓** | Scrolls down one page. |
| **Page down** | Scrolls down one page. |

On some platforms, you may need to adjust your X Window System keyboard mapping to use some of the keys on your nume+ric keypad. Refer to Appendix B, "*Operating Systems*" on page 329 for details.

# Scrolling Multiline Fields

You can scroll multiline fields in dialog boxes. The bottom left corner of the multiline field indicates your location in the field as follows:

| | |
|---|---|
| All | All lines are visible. |
| Top | The top-most lines are visible, and more lines are below the bottom of the field. |
| Bot | The bottom-most lines are visible and more lines are above the top of the field. |
| *nn*% | The percentage of the lines above the top of the field that are not visible is indicated. |

The following figure shows an example of a scrollable multiline field.

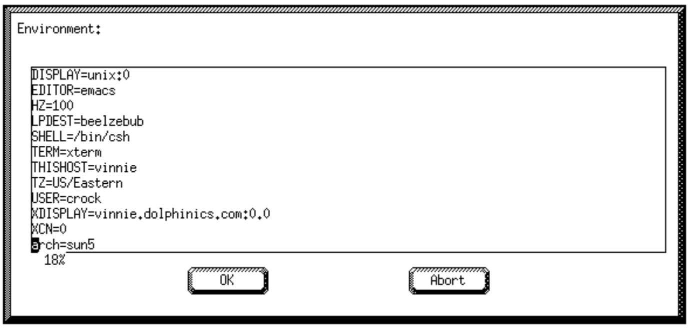



**Figure 12: Scrollable Multiline Field**

You can use the following keys to move within a multiline field:

| | |
|---|---|
| ↑ or **Ctrl-P** | Moves up a line. |
| ↓ or **Ctrl-N** | Moves down a line. |
| **Ctrl-U** | Multiplies one of the previous key commands by four. For example, if you enter **Ctrl-U Ctrl-P**, the cursor moves up four lines. |

When you move off the top or bottom of the field, the field scrolls automatically by one line.

# Diving into Objects

To display more detail about an object (for example, a variable), place the cursor over the object and dive into it by clicking the Dive mouse button (which is usually the right button). You can dive into any object that has a block of data associated with it, such as a pointer, structure, or subroutine. TotalView displays the information about the object in the current window or in a separate window, as described in Table 3.

TABLE 3: Uses for Diving

| Dive on: | Information Displayed by Diving: |
|---|---|
| Process or thread | A Process Window appears focused on a thread |
| Routine in the Stack Trace Pane | The stack frame and source code for the routine appear in the Process Window |
| Pointer | The referenced memory area appears in a separate Variable Window |
| Variable | The contents of the variable appear in a separate Variable Window |
| Array element, structure element, or referenced memory area | The contents of the element or memory area replaces the contents that was in the Variable Window—this is known as a *nested* dive |
| Subroutine | The source code for the subroutine appears in the Process Window<br><br>A subroutine must be compiled with source line information (usually, with the **–g** option) for you to dive into it and see source code; if the subroutine was not compiled with source line information, TotalView displays the assembler code for the routine |

For additional information about displaying variable contents, refer to "D*iving in Variable* Windows" on page 147.

# Editing Text

The TotalView field editor lets you change the values of fields in windows or to change text fields in dialogs. To edit text:

**1** Click the left mouse button to select the text you wish to change. If you can edit the selected text, it appears within a rectangle, and you will see an **editing cursor** (a black rectangle).

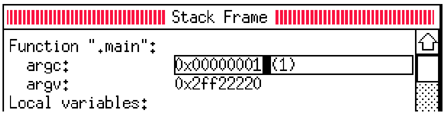



**Figure 13: Editing Cursor**

**2** Edit the text and press Return (for single-line fields) or Shift-Return (for multiline fields).

You can copy and paste text within TotalView windows, between TotalView windows, or between TotalView windows and other X Window System windows.

The following steps explain how you copy and paste text between an editable field in TotalView and other X Windows applications. Note that this procedure is unique to TotalView.

**1** Copy text into the cut buffer with one of the following:

➤ Clicking and holding the left mouse button at one end of the range, dragging the cursor to the other end of the range, then letting go of the mouse button, or

➤ Clicking the left mouse button at one end of the range then right clicking the mouse button at the other end of the range

TotalView highlights the text while you hold the mouse button down. When you release the mouse button, the highlight disappears indicating TotalView copied the text into the cut buffer.

**2** Move the cursor to where you want to paste the text, then either:

➤ Press Control middle mouse button, or

➤ Press the middle mouse button for a menu. Select **Paste (Ctrl-V)** from the menu.

The following table describes the field editor commands. Many of these commands perform the same operation in the Emacs text editor.

TABLE 4: **Field Editor Commands**

| Keystrokes | Action |
|---|---|
| Ctrl-A | Moves the cursor to the beginning of the line |
| Ctrl-B | Moves the cursor backward one character |
| Ctrl-C | Aborts the field editor, and discard all changes |
| Ctrl-D | Deletes the character under the cursor |
| Ctrl-E | Moves the cursor to the end of the line |
| Ctrl-F | Moves the cursor forward one character |
| Ctrl-H, Backspace, or Delete | Deletes the previous character |
| Ctrl-K | Deletes all text to the end of the line, or deletes a newline |
| Ctrl-N | Moves the cursor to the next line in a multiline control |
| Ctrl-O | Inserts a newline in a multiline control |
| Ctrl-P | Moves the cursor to the previous line in a multiline control |
| Ctrl-U [*n*] | Multiplies the number of times a command is executed by *n*; *n* is optional; the default is 4<br>Use this command in combination with another command; for example, to move the cursor forward 50 characters, type Ctrl-U 50 Ctrl-F |
| Ctrl-V | Pastes text from the X Windows copy buffer |
| Return | For single-line fields, stops the field editor and deselects the field<br>In dialog boxes, confirms the dialog box as if you had selected the **OK**, **Continue**, or **Yes** button<br>For multi-line fields, inserts a newline |
| Shift-Return | For both single-line and multi-line fields, stops the field editor and deselects the field<br>In dialog boxes, confirms the dialog box as if you had selected the **OK**, **Continue**, or **Yes** button |

TABLE 4: **Field Editor Commands (cont.)**

| Keystrokes | Action |
| --- | --- |
| Tab | Spaces over to the next tab stop—tab stops are located every four characters |
| ↑, ↓, ←, → | Moves up, down, backward, and forward one character |

## Searching for Text

You can search for text strings in most TotalView windows, as follows:

- **Search for String**

  Searches forward in the window for a text string. The debugger prompts you for the string. The search starts from the first line of text that is visible in the window.

- **Search Backward for String**

  Searches backward for a text string. The search starts from the last line of text that is visible in the window.

- **Reexecute Last Search**

  Repeats the last forward or backward search without prompting for a string. The search starts from the point where the last search left off and continues in the same direction.

## Using the Spelling Corrector

TotalView can check the spelling of text entries for certain commands. If TotalView does not find the name you entered, it displays a dialog box with the closest match, as shown in Figure 14.

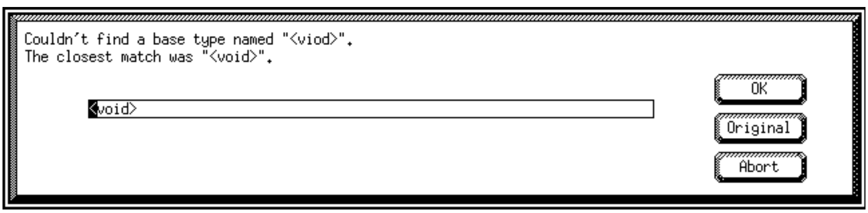



**Figure 14: Spelling Corrector Dialog Box**

You can edit the closest match, and then select **OK** to use it, **Original** to get back the original text, or **Abort** to cancel.

To customize the behavior of the spelling corrector with X Window System resources, refer to "TOTALVIEW*SPELLCORRECTION" on page 291.

## Saving the Contents of Windows

You can save the contents of most window panes as ASCII text. You can:

- **Write data to a file**

  When you specify *filename*, TotalView checks to see if the file exists. If it exists, the debugger overwrites the file's contents. If it does not exist, TotalView creates the file, then writes the information to it.

- **Append data to a file**

  When you specify +*filename*, TotalView checks to see if the file exists. If it exists, TotalView appends information to the end of the file. If it does not exist, TotalView creates the file, then writes the information to it.

- **Pipe data to UNIX shell commands**

  When you specify |*command*, TotalView pipes the commands to **/bin/sh** for execution. You can use a series of shell commands if desired. For example, here is a command that ignores the top five lines of output, compares the current ASCII text to an existing file, and writes the differences to another file:

  | tail +5 | diff – *filename* > *filename.diff*

Here is the procedure for saving the contents of the current window pane:

1. Move the mouse pointer into the desired pane and select the **Save Window to File** command from the menu.
2. Enter *filename*, +*filename*, or | *command* in the dialog box and then select **OK**.

To save a series of panes in a window, you can use the **Reexecute Last Save Window** command. This command repeats the last **Save Window to File** command (including the information entered in the dialog box).

# Exiting from TotalView

You can exit from TotalView either by pressing **Ctrl-Q** in a window or by selecting the **Quit Debugger** command in the Root Window.

After entering one of these commands, TotalView displays a dialog. Select **Yes** or type **y** to confirm. If you do not want to exit, select **No** or type **n**. As TotalView exits, it kills all programs and processes that it started. However, programs and processes used while you were debugging your program and which it did not start continue to execute.

Chapter 3

# Setting Up a Debugging Session

This chapter explains how to set up basic TotalView sessions. It also describes some common commands and procedures. For information on setting up remote debugging sessions, see Chapter 4, "*Setting Up Remote Debugging Sessions*" on page 51. For information on setting up parallel debugging sessions, see Chapter 5, "*Setting Up Parallel Debugging Sessions*" on page 69.

In this chapter, you will learn how to:

- Compile programs
- Start TotalView
- Load executables
- Attach to and detach from processes
- Examine core files
- Determine the status of processes and threads
- Handle signals
- Set search paths
- Set command arguments and environment variables
- Set input and output files
- Monitor your TotalView session

## Compiling Programs

Before you start to debug a program, you must compile the program with the appropriate options and libraries for your situation. Table 5 presents some general considerations, but you must check Appendix A, "*Compilers and Environments*," on page 319 to determine the exact syntax and any other considerations for your platform. For additional information on how to compile a Portland Group HPF program for debugging, see "*Compiling HPF for Debugging*" on page 109.

TABLE 5: Compiler Considerations

| Compiler Option or Library | What It Does | When to Use It |
|---|---|---|
| Debugging symbols option (usually **–g**) | Generates debugging information in the symbol table | Before debugging *any* program with TotalView |
| Optimization option (usually **–O**) | Moves code to optimize execution of program<br>Some compilers do not let you use the **–O** option with the **–g** option<br>Even if you can, we recommend against it because using the **–O** option when debugging your program can produce strange results | After you finish debugging your program with TotalView |
| Multiprocess programming library (usually **dbfork**) | Uses special versions of the **fork()** and **execve()** system calls<br>Using dbfork is discussed in "*Linking with the dbfork Library*" on page 324 | Before debugging a multiprocess program that explicitly calls **fork()** or **execve()**<br>Refer to "*Processes That Call fork()*" on page 205 and "*Processes That Call execve()*" on page 205 |

# Starting the TotalView Debugger

The command syntax for starting TotalView is:

**totalview** [ *executable* [ *corefile* ] ] [ *options* ]

where *executable* specifies the name of the executable file to be debugged and *corefile* specifies the name of the core file to be debugged.

**NOTE** If you are starting the CLI, you will type "totalviewcli" rather than "totalview".

Here are some of the common ways to start TotalView:

**totalview**
Starts TotalView without loading a program or core file. After TotalView starts, you can load a program by using the **New Program Window** command from the Root Window.

**totalview** *executable*
Starts TotalView and loads the *executable* program.

**totalview** *executable corefile*
Starts TotalView and loads the *executable* program and the *corefile* core file.

**totalview** *executable* **–a** *args*
Starts TotalView and passes all subsequent arguments (specified by *args*) to the *executable* program. If you use the **–a** option, it must appear after all other TotalView options on the command line.

**totalview** *executable* **–grab**
Starts TotalView and grabs the keyboard whenever it displays a dialog box. You should use this option whenever you start TotalView with a window manager that uses a "click-to-type" model.

**totalview** *executable* **–remote** *hostname*[:*portnumber*]
Starts TotalView on the local host and the TotalView Debugger Server (**tvdsvr**) on the remote host *hostname*. Loads the program specified by *executable* for remote debugging. You can specify a host name or TCP/IP address for *hostname*, and optionally, a TCP/IP port number for *portnumber*.

For more information on:

- Debugging parallel programs such as MPI, PVM, or HPF, refer to Chapter 5, "*Setting Up Parallel Debugging Sessions*" on page 69.
- The **totalview** command, refer to Chapter 12, "*TotalView Command Syntax*" on page 299.
- Remote debugging, refer to "*Debugging Remote Processes*" on page 51, "*Starting the Debugger Server for Remote Debugging*" on page 55, and Chapter 13, "*TotalView Debugger Server Command Syntax*" on page 311.

# Loading Executables

## Loading a New Executable

If you did not load an executable when starting TotalView, you can load one using the **New Program Window** command as follows:

**1** From the Root Window, select the **New Program Window** command. The following dialog box appears.

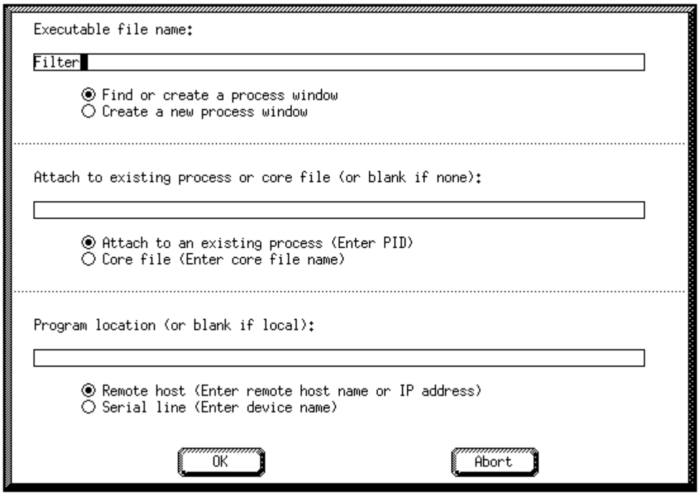



FIGURE 15: **New Program Window Dialog Box**

**2** Enter the name of the executable in the **Executable file name** field. You can use a full or relative pathname.

If you enter a simple filename, TotalView searches for it in the list of directories specified with the **Set Search Directory** command or named in your **PATH** environment variable.

**3** To create a new process instead of reusing an existing one, select the **Create a new process window** button. Afterwards, TotalView adds an entry in the Root Window for the process.

**4** Select **OK**.

If you use the **New Program Window** command to reload the current executable, TotalView does not reread the executable; instead, it reuses the existing symbol table. To have TotalView reread the executable, you need to use the **Reload Executable File** command, as described in the next section.

### Reloading a Recompiled Executable

If you edit and recompile your program during a debugging session, you can reload the updated program without exiting from TotalView, as follows:

1. Confirm that all processes using the executable have exited. If they have not, display the **Arguments/Create/Signal** menu and select the **Delete Program** command from the Process Window.
2. Confirm that duplicate copies of the process do not exist by entering the **ps** shell command. If duplicate processes exist, delete them with the **kill** command.
3. Recompile your program.
4. In the Process Window, display the **Arguments/Create/Signal** menu and select the **Reload Executable File** command. TotalView updates the Process Window with the new source file and loads the new executable file.

   The next time you start this process, TotalView uses the new executable file.

## Attaching to Processes

If a program you are testing is hung or looping (or misbehaving in some other way), you can attach to it with TotalView. You can attach to single processes, multiprocess programs, and remote processes.

To attach to a process, you can either use the **Show All Unattached Processes** or **New Program Window** commands.

If the process or any of its children calls the **execve()** routine, you may need to attach to it by creating a new program window. This is because TotalView uses the **ps** command on some platforms to obtain the name of the process executable. Since **ps** can give incorrect names, TotalView may not find it.

**NOTE** When you exit from TotalView, it kills all programs and processes that it started. However, programs and processes used while you were debugging your program and which it did not start continue to execute.

## Attaching Using Show All Unattached Processes

To attach to a process using the **Show All Unattached Processes** command, go to the Root Window and complete the following steps:

**1** Select the **Show All Unattached Processes** command.

The **Processes that TotalView doesn't own** Window appears, as shown in Figure 16. This window lists the process ID, status, and name of each process associated with your username. The processes that appear dimmed are those that are being debugged or those that TotalView will not allow you to debug (for example, the TotalView process itself).

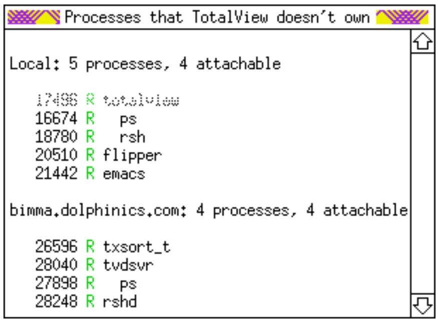



FIGURE 16: **Processes that TotalView doesn't own Window**

The processes at the top of this figure are all local. The remaining processes are remote.

If you are debugging a remote process, this window also shows processes running under your username on each remote host name. You can attach to any of these remote processes. For more information on remote debugging, refer to "*Starting the Debugger Server for Remote Debugging*" on page 55 and Chapter 13, "*TotalView Debugger Server Command Syntax*" on page 311.

**2** Dive into the process you wish to debug.

A Process Window appears. The right arrow points to the current program counter (PC), which indicates where the program is executing or where it is hung.

## Attaching Using the New Program Window

To attach to a process by using the **New Program Window** command, follow these steps:

**1** Use the **ps** shell command to obtain the process ID (PID) of the process.

**2** Issue the **New Program Window** command from the Root Window. TotalView displays the following figure.

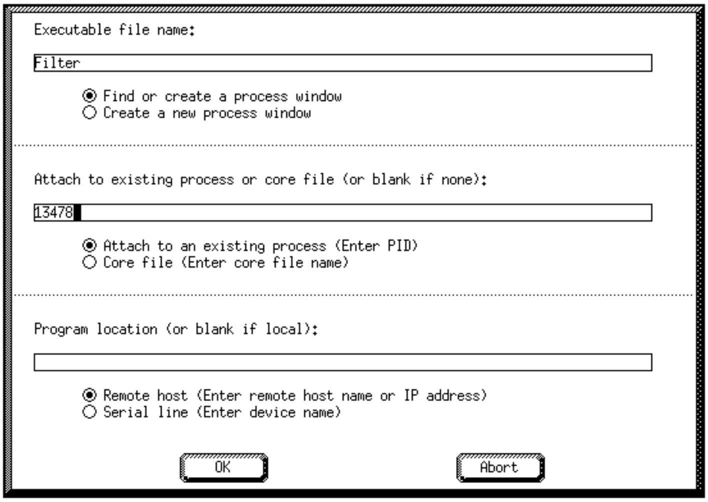



FIGURE 17: **New Program Window Dialog Box**

Enter a file name in the **Executable field name** field. This name can be a full or relative pathname. If you supply a simple filename, TotalView searches for it in the directories specified with the **Set Search Directory** command and specified by your **PATH** environment variable.

Enter the process ID (PID) of the *unattached* process in the middle section of the dialog box.

**3** Select **OK**.

If the executable is a multiprocess program, TotalView asks if you want to attach to all relatives of the process. To examine all processes, select **Yes**.

If the process has children that called **execve()**, TotalView tries to determine the correct executable for each of them. If TotalView cannot determine the executables for the children, you need to delete (kill) the parent process and start it again using TotalView.

Finally, a Process Window appears. The right arrow points to the current program counter (PC), which is where the program is executing or where it is hung.

# Detaching from Processes

You can detach from processes that TotalView did not create when you finish debugging them using the following procedure:

1. If you want to send the process a signal, select the **Set Continuation Signal** command. Choose the signal that TotalView should send to the process when it detaches from the process. For example, to detach from a process and leave it stopped, set the continuation signal to SIGSTOP.
2. Display the **Arguments/Create/Signal** menu and select the **Detach from Process** command.

When you detach from a process, TotalView removes all breakpoints that were set within it.

# Examining a Core File

If a process encounters a serious error and dumps a core file, you can examine it using one of the following methods:

- Start TotalView as follows:

  **totalview** *filename corefile* [ *options* ]
- Enter the **New Program Window** command from the Root Window. In the middle section of the dialog box, enter the name of the core file, select the **Core file** radio button, and then select **OK**.

**NOTE** You can only debug local core files.

The Process Window displays the core file, with the Stack Trace, Stack Frame, and Source Code Panes showing the state of the process when it dumped core. The title bar of the Process Window names the signal that caused the core dump. The right arrow in the tag field of the Source Code Pane indicates the value of the program counter (PC) when the process encountered the error.

You can examine the state of all variables at the time the error occurred. Chapter 7, "*Examining and Changing Data*" on page 143 contains more information.

If you start a process while you are examining a core file, TotalView stops using the core file and starts a fresh process with the executable.

# Determining the Status of Processes and Threads

Process and thread states are displayed in:

- The Root Window, for processes and threads.
- The Unattached Processes Window, for processes.
- The process and thread status bars of the Process Window.
- The Thread List Pane of the Process Window, for threads.

## Process Status

The status of a process includes three elements: the process location, the process ID, and the state of the process. The Root Window displays a single character to identify the state of a process. (These characters are explained in "*Attached Process States*" on page 40.) The process status in the Root Window has the following form:

[ *L* ] *N S process_name*

where:

| | |
|---|---|
| *L* | The process location (present only for remote processes) |
| *N* | The process ID |
| *S* | The single-character representation of the process state |
| *process_name* | TotalView's name for the process |

The **Unattached Process** Window lists all processes associated with your username. The information in this window is similar to the information in the

Root Window; process states are specified with a single character. Processes being debugged are dimmed out.

The process status bar of the Process Window displays information in the following format:

**Process [ *L* ] *N*:** *process_name* **(** *state* **)**

where:

| | |
|---|---|
| L | The process location (remote processes only) |
| N | The process ID |
| *process_name* | TotalView's name for the process |
| *state* | The state name of the process based on the state of its threads |

## Thread Status

The Root Window displays a single character that identifies the state of a thread. (These characters are explained in "*Attached Process States*" on page 40.) The thread status in the Root Window has the following form:

*T* / *X S* in *routine_name*

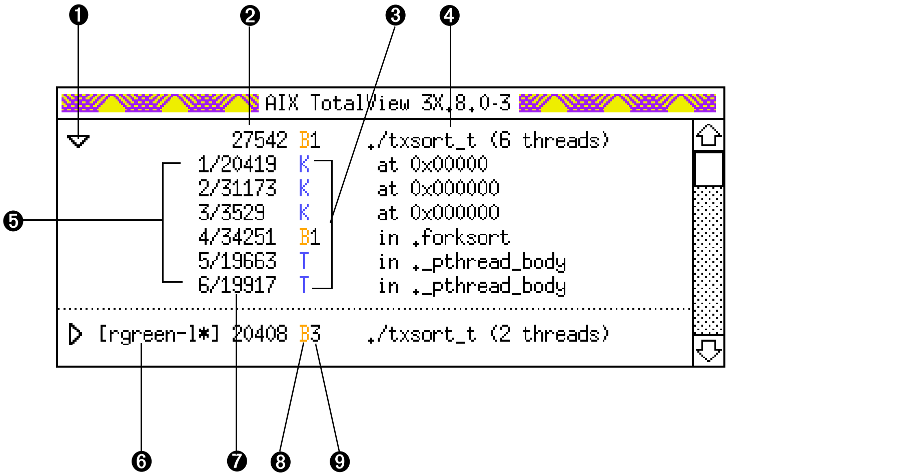
where:

| | |
|---|---|
| *T* | The TotalView assigned thread ID |
| X | The system assigned thread ID |
| S | The single-character representation of the thread's state |
| *routine_name* | The name of the routine in which the thread was executing when last stopped by TotalView |

Figure 18 shows process and thread status.

If, as they are on some systems, the TotalView-assigned thread ID and the system-assigned thread ID are the same, TotalView displays only one ID value.

The Thread List Pane in the Process Window uses the same thread status format as the Root Window.




❶ Collapse/expand toggle
❷ Process ID (pid)
❸ Thread status
❹ Program name
❺ Thread list
❻ Remote process location
❼ Thread ID (tid/systid)
❽ Process status
❾ Action point ID number

FIGURE 18: **Root Window Showing Process and Thread Status**

The thread status bar of the Process Window displays information in the following format:

Thread *N*.*T*: *process_name* (*state*) *reason*

where

| | |
|---|---|
| *N* | The process ID |
| *T* | The TotalView assigned thread ID |
| *process_name* | The TotalView's name for the process |
| *state* | The state name of the thread |
| *reason* | The reason the thread stopped |

## Unattached Process States

The state information for a process displayed in the Unattached Processes Window is derived from the system. The state characters TotalView uses to summarize the state of an unattached process do not necessarily match those used by the system.

Table 6 summarizes the possible states in the Unattached Processes Window.

TABLE 6: Summary of Unattached Process States

| State | State Character | Meaning for a process |
|---|---|---|
| Running | R | Process is running or can run |
| Stopped | T | Process is stopped |
| Idle | I | Process has been idle or sleeping for *more* than 20 seconds |
| Sleeping | S | Process has been idle or sleeping for *less* than 20 seconds |
| Zombie | Z | Process is a "zombie"; that is, a child process that has terminated and is waiting for its parent process to gather its status |

## Attached Process States

The state of processes and threads that TotalView is attached to is displayed in various windows.

Table 7 summarizes the possible states for an attached process or thread, and how these states are displayed.

TABLE 7: Summary of Attached Process and Thread States

| State Name | State Character | Meaning for a thread and process |
|---|---|---|
| At breakpoint | B | *Thread*: stopped at a breakpoint<br>*Process*: one or more threads are stopped at a breakpoint |
| Error *reason* | E | *Thread*: is stopped because of error *reason*<br>*Process*: one or more threads are in the **Error** state |
| In kernel | K | *Thread only*: the thread is executing inside the kernel (that is, it made a system call); when a thread is in the kernel, the operating system does not allow TotalView to view the full state of the thread |

TABLE 7: Summary of Attached Process and Thread States (cont.)

| State Name | State Character | Meaning for a thread and process |
|---|---|---|
| Running | R | *Thread*: is running or can run<br>*Process*: all threads in the process are running or can run |
| Exited or never created | Blank | *Process only*: does not exist |
| Mixed | M | *Process only*: some threads in the process are running and some are not running or the process is expecting some of its threads to stop |
| Stopped *reason* | T | *Thread*: stopped because of *reason*, but not at a breakpoint and not because of an error<br>*Process*: one or more threads are stopped, but none are in the **At Breakpoint** state |
| At watchpoint | W | *Thread*: stopped at a watchpoint<br>*Process*: one or more threads are stopped at a watch point |

The **Error** state usually indicates that your program received a fatal signal from the operating system. Signals such as SIGSEGV, SIGBUS, and SIGFPE can indicate an error in your program. The next section shows the procedure for controlling how TotalView handles signals that your program receives.

# Handling Signals

If your program contains a signal handler routine, you may need to adjust the way TotalView handles signals. You can do this by using:

- A dialog box (described in this section)
- An X resource (see "TOTALVIEW*SIGNALHANDLINGMODE" on page 290)
- A command-line option to the **totalview** command (refer to "TOTALVIEW COMMAND SYNTAX" on page 299)

Unless you tell it otherwise, here is how TotalView handles UNIX signals:

TABLE 8: **Default Signal Handling Behavior**

| Signals that are Passed Back to Your Program | Signals that Stop Your Program or Cause an Error |
|---|---|
| SIGHUP | SIGILL |
| SIGINT | SIGTRAP |
| SIGQUIT | SIGIOT |
| SIGKILL | SIGEMT |
| SIGALRM | SIGFPE |
| SIGURG | SIGBUS |
| SIGCONT | SIGSEGV |
| SIGCHLD | SIGSYS |
| SIGIO | SIGPIPE |
| SIGVTALRM | SIGTERM |
| SIGPROF | SIGTSTP |
| SIGWINCH | SIGTTIN |
| SIGLOST | SIGTTOU |
| SIGUSR1 | SIGXCPU |
| SIGUSR2 | SIGXFSZ |

TotalView uses the SIGTRAP and SIGSTOP signals internally. If the process encounters either signal, TotalView neither stops the process with an error nor passes the signal back to your program. Further, you cannot alter the way TotalView uses these signals.

On some systems, hardware registers can affect how signals such as SIGFPE are handled. For more information, refer to "*Interpreting Status and Control Registers*" on page 122 and Appendix C, "*Architectures*," on page 343.

**NOTE** On SGI machines, setting the TRAP_FPE environment variable to any value indicates that your program will trap underflow errors. If you set this variable, however, you will also need to use this dialog box to tell TotalView what it should do with SIGFPE errors. (In most cases, you will set SIGFPE to *Resend*.) As alternatives, you can set the -signalHandlingMode option (see page 309) or the totalview*signalHandlingMode X resource (see page 290) to "Resend=SIGFPE".

You can change the signal handling mode by going to the Process Window and display the **Arguments/Create/Signal** menu. You then select the **Set Signal Handling Mode...** command. The dialog box shown in Figure 19 appears.

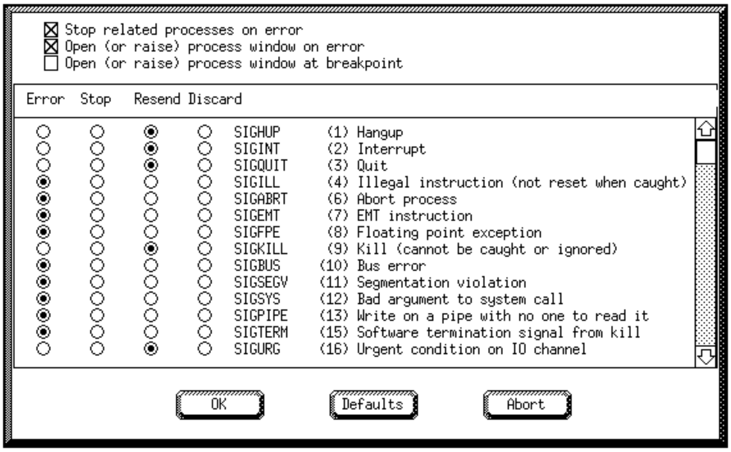



FIGURE 19: **Set Handling Mode Command Dialog Box**

**NOTE** The signal names and numbers shown in the dialog box are platform-specific.

When your program encounters an error signal, TotalView stops all related processes. If you do not want this behavior, deselect the **Stop related processes on error** checkbox.

Also by default, when your program encounters an error signal, TotalView opens or raises the Process Window. Deselecting the **Open (or raise) Process window on error** checkbox tells TotalView that it shouldn't open or raise the window. You can change the default setting of this checkbox using an X resource ("TOTALVIEW*POPONERROR" on page 287) or a command line option.

If a processes in a multiprocess program encounter an error, TotalView only opens a Process Window for the *first* process that encounters an error. This feature prevents the screen from filling up with process windows.

If you select the **Open (or raise) process window at breakpoint** checkbox, TotalView opens or raises the Process Window when your program reaches a breakpoint. You can make this behavior your default by using an X Resource ("TOTALVIEW*POPATBREAKPOINT" on page 287) or a command line option.

If necessary, scroll the signal list to the signal being changed. Make your changes by selecting one of the following radio buttons:

**Error** Stops the process, places it in the error state, and displays an error in the title bar of the Process Window. If the **Stop related processes on error** checkbox is selected, TotalView also stops all related processes. You should select this signal handling mode for severe error conditions such as SIGSEGV and SIGBUS signals.

**Stop** Stops the process and places it in the stopped state. Select this signal handling mode if you want TotalView to handle this signal the same as a SIGSTOP signal.

**Resend** Sends the signal to the process. If your program contains a signal handling routine, you should use this mode for all the signals that it handles. By default, the common signals for terminating a process (SIGKILL and SIGHUP) use this mode.

**Discard** Discards the signal and restarts the process without a signal.

**NOTE** Do not use Discard mode for fatal signals, such as SIGSEGV and SIGBUS. If you do, TotalView can get caught in a signal/resignal loop with your program; the signal will immediately recur because the failing instruction will reexcute repeatedly.

# Setting Search Paths

If your source code, executable, or object files reside in different directories, set search paths for these directories with the **Set Search Directory** command. TotalView searches the following directories (in order):

1. The current working directory (**.**).
2. The directories you specify with the **Set Search Directory** command in the exact order you enter them in the dialog box.
3. If you specified a full pathname for the executable when you started TotalView, TotalView searches this directory.
4. The directories specified in your PATH environment variable.

These search paths apply to *all* processes that you are debugging.

To use the **Set Search Directory** command, go to the Process Window and complete these steps:

1 Display the **Display/Directory/Edit** menu and select the **Set Search Directory…** command. The following dialog box appears.

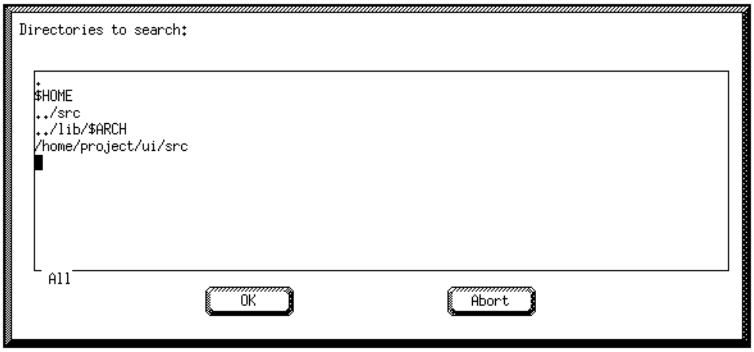



FIGURE 20: **Set Search Directory Dialog Box**

2 Enter the directories in the order you want them searched, separating each directory with a space. You can also enter them on separate lines.

The current working directory (**.**) is the first directory listed in the window. You can move the current working directory further down the list. If you remove it, TotalView reinserts it at the top of the list. Relative pathnames are interpreted as being *relative* to the current working directory.

3 Select **OK**.

Once you change the list of directories with the **Set Search Directory** command, TotalView again searches for the source file that is currently displayed in the Process Window.

You can also specify search directories using an X Window System resource. Refer to "TOTALVIEW*SEARCHPATH" on page 289.

# Setting Command Arguments

When TotalView creates a process, it passes the name of the file containing the executable code for the process to the program. If your program requires command line arguments, you must set these arguments *before* you start the process, as follows:

**1** Display the **Arguments/Create/Signal** menu and select the **Set Command Arguments...** command. The following dialog box appears:

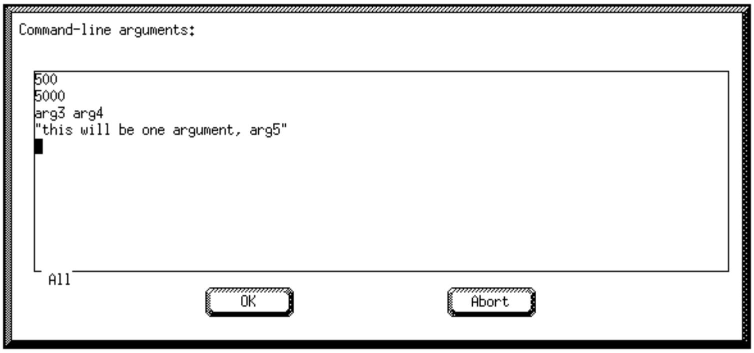



FIGURE 21: **Set Command Arguments Dialog Box**

**2** Enter the arguments to be passed to the program. Separate each argument with a space, or place each argument on a separate line. If an argument has spaces in it, enclose the whole argument in double quotes. When you are done, select **OK**.

You can also set command-line arguments with the **–a** option of the **totalview** command, as discussed in "*Starting the TotalView Debugger*" on page 30.

# Setting Environment Variables

You can set and edit the environment variables that TotalView passes to processes. When TotalView creates a new process, it passes a list of environment variables to the process. By default, a new process inherits TotalView's environment variables, and a remote process inherits **tvdsvr**'s environment variables.

If the Environment Variable dialog is empty, the process inherits its environment variables from TotalView or **tvdsvr**.

**NOTE If you add environment variables, the process no longer inherits environment variables; it only receives the variables that you enter in this dialog box. Therefore, if you want to add additional variables to those inherited that would be inherited, you must enter the variables being inherited in addition to the ones you are adding.**

An environment variable is specified as *name=value*. For example, the following definition creates an environment variable named **DISPLAY** whose value is **unix:0.0**:

DISPLAY=unix:0.0

To add, delete, or modify environment variables, go to the Process Window and display the **Arguments/Create/Signal** menu and select the **Set Environment Variables** command. In the displayed dialog box, place each environment variable on a separate line. TotalView ignores blank lines. Figure 22 shows an example:

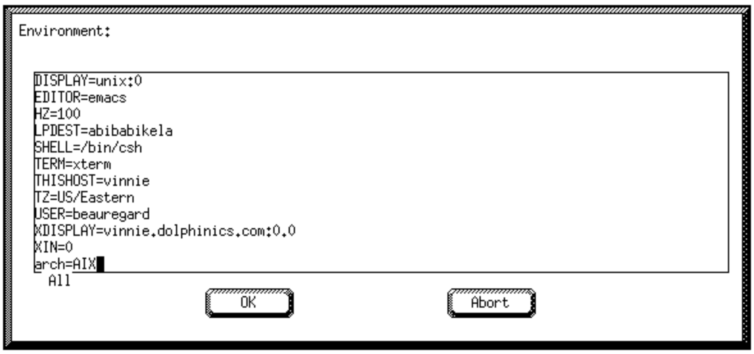



FIGURE 22: **Environment Variables Dialog Box**

The actions you can now perform are:

- To change the name or value of an environment variable, edit the line.
- To add a new environment variable, insert a new line and specify the name and value.

- To delete an environment variable, delete the line. If you delete all the lines, the process inherits TotalView or **tvdsvr**'s environment.

## Setting Input and Output Files

Before TotalView begins executing a program, it determines how it will handle standard input (**stdin**) and standard output (**stdout**). Unless you tell it otherwise, **stdin** and **stdout** use the shell window from which TotalView was invoked.

You can redirect **stdin** or **stdout** to a file by completing these steps from the Process Window before you start executing your program:

**1** Display the **Arguments/Create/Signal** menu and select either **Input from File...** or **Output to File...** . The following dialog box appears.

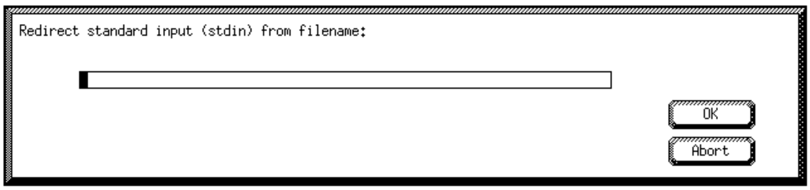



FIGURE 23: **Input from File Dialog Box**

**2** Enter the name of the file, relative to your current working directory.

**3** Select **OK**.

## Monitoring TotalView Sessions

TotalView logs all significant events occurring for all processes being debugged. To view the event log, go to the Root Window and select the **Show Event Log Window** command. The TotalView Event Log Window displays a sequential list of events. For example:

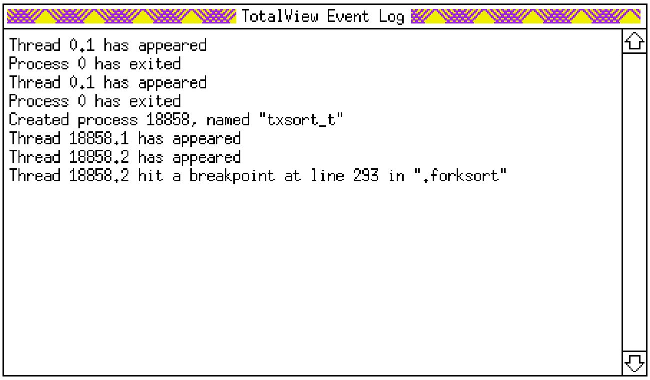



FIGURE 24: **Event Log Window**

Chapter 4

# Setting Up Remote Debugging Sessions

This chapter explains how to set up TotalView remote debugging sessions. In this chapter, you will learn how to:

- Debug remote processes
- Connect to remote machines
- Start the debugger server for remote debugging
- Launching programs using a large number of computers (bulk server launch)
- Debug over a serial line

## Debugging Remote Processes

You can begin debugging remote processes either by loading a remote executable or by attaching to a remote process.

**NOTE** You cannot examine core files on remote nodes.

### Loading a Remote Executable

Here is the procedure for loading a remote program into TotalView:

1. Select the **New Program Window** command (see "*Loading a New Executable*" on page 32 for more information) and then enter the executable's file name and select the **Create a new process window** button.
2. Enter the host name or TCP/IP address of the machine on which the executable should be running in the **Program location field**, as shown in Figure 25.

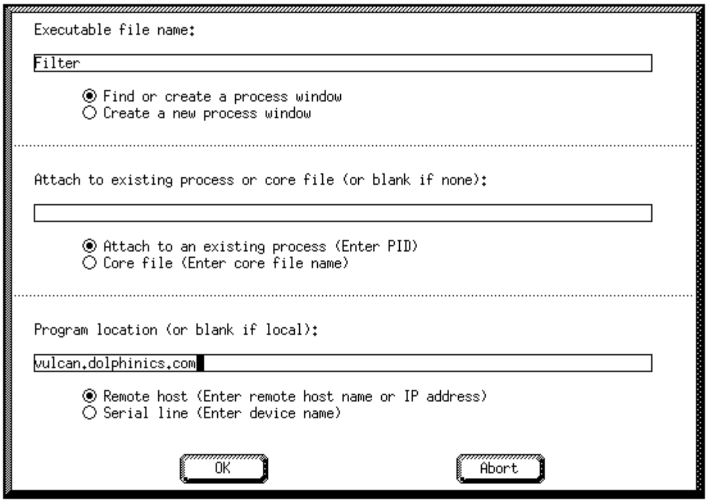



FIGURE 25: **New Program Window Dialog Box**

On some multiprocessor platforms, TotalView displays additional radio buttons in the lower section of the dialog box. You can use these buttons for debugging programs that are running on groups or clusters of processors.

**3** Select OK.

If this method does not work, you may need to disable the auto-launch feature for this connection and start the TotalView Debugger Server (**tvdsvr**) manually. Then, you can specify *hostname*:*portnumber* in step 2, where *portnumber* is the TCP/IP port number on which the debugger server is communicating with TotalView. For more information on this alternative, refer to "*Starting the Debugger Server for Remote Debugging*" on page 55.

## Attaching to a Remote Process

You attach to a remote process using the same dialog boxes you use when you attach to a local process. You will, however, enter information in different fields. You can also attach to a remote process by bringing up the associated windows, then diving into processes from them.

Here is how you attach to a remote process:

1 After using the **ps** shell command to obtain the process ID, display the **New Program Window**. (See "*Attaching Using the New Program Window*" on page 35 for more information.)
2 Enter a file name in the **Executable field name** field and the process ID in the **Attach to ...** field.
3 Enter the host name or TCP/IP address of the machine on which the executable should be running in the bottom section of the dialog box.

   On some multiprocessor platforms, TotalView displays additional radio buttons in the lower section of the dialog box. You can use these buttons for debugging programs that are running on groups or clusters of processors.
4 Select **OK**.

If this method does not work, you may need to disable the auto-launch feature for this connection and manually start the debugger server. You can now specify *hostname*:*portnumber* in step 2 where *portnumber* is the TCP/IP port number on which **tvdsvr** is communicating with TotalView. For more information on this alternative, refer to "*Starting the Debugger Server for Remote Debugging*" on page 55.

You can also attach to a remote process by first connecting to a remote host using the **New Program Window** command and then bringing up a list of unattached processes with the **Show All Unattached Processes** command. You can attach to these processes by diving into them.

1 Connect to the remote host. For details, see "*Connecting to Remote Machines*" on page 53.
2 After connecting to the remote host, bring up a list of unattached processes. You can attach to these processes by diving into them. For details, see "*Attaching Using Show All Unattached Processes*" on page 34.

# Connecting to Remote Machines

You can connect to a remote machine in two ways: with the **–remote** option on the command line when you start TotalView or with the **New Program Window** command from the Root Window after you start TotalView.

**NOTE** If TotalView supports a parallel process runtime library (for example, MPI, PVM, or HPF), it automatically connects to remote hosts. For more information, see Chapter 5 "*Setting Up Parallel Debugging Sessions*" on page 69.

For details on the syntax for the **–remote** command-line option, see "*Starting the TotalView Debugger*" on page 30.

To connect to a remote host from a TotalView session, follow these steps:

**1** Issue the **New Program Window** command from the Root Window. A dialog box appears, as shown in the following figure.

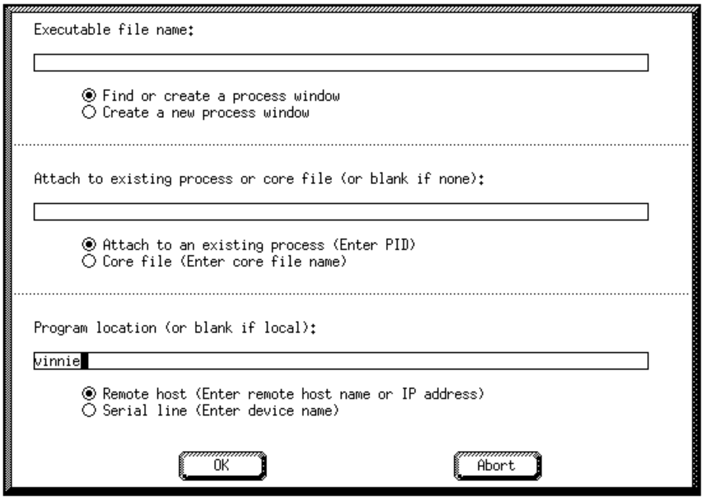



FIGURE 26: **Remote Host Connection**

**2** Delete the text from the **Executable file name** and **Attach to existing process or core file** fields.

**3** Enter the host name or TCP/IP address of the machine on which the executable will be running in the bottom section of the dialog box.

On some multiprocessor platforms, TotalView displays additional radio buttons in the lower section of the dialog box. These buttons let you debug programs running on groups or clusters of processors.

**4** Select **OK**.

# Starting the Debugger Server for Remote Debugging

Debugging a remote process with TotalView only differs from debugging a native process in that:

- TotalView works with another TotalView process running on the remote machine to debug the remote process. This process is called the TotalView Debugger Server (**tvdsvr**)
- The performance of your session depends on the performance of the network between the native and remote machines. If the network is overloaded, debugging can be slow.

Unless you tell it otherwise, TotalView automatically launches **tvdsvr**. It can be launched in two ways. The first method launches a **tvdsvr** on each remote host independently. The second method, called *bulk server launch*, launches all remote processes at the same time. Auto-launching greatly simplifies the debugging remote processes since you do not need to take any action to debug remote processes.

## Single Process Server Launch Options

The **Server Launch Window** dialog box lets you change the server launch command, disable auto-launch, and alter the connection timeout used by TotalView when it launches **tvdsvr**.

The popup menu in the Root Window contains the **Server Launch Window** command. After selecting this command, TotalView displays the following dialog box:

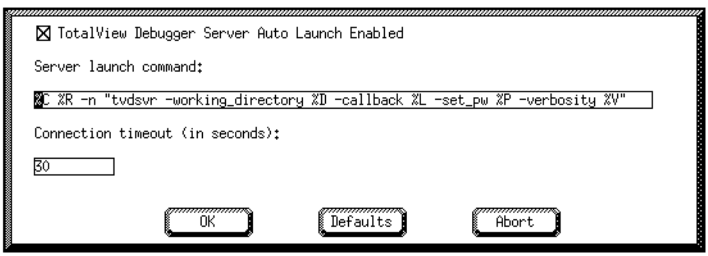



FIGURE 27: **Server Launch Window**

**TotalView Debugger Server Auto Launch Enabled**
If the checkbox is selected, TotalView will auto-launch the TotalView Debugger Server (**tvdsvr**).

**Server launch command**
If auto-launch is enabled, TotalView will use this command to launch **tvdsvr**. For information on this command and its options, see "*Single Process Server Launch Command*" on page 59.

**Connection timeout**
After TotalView automatically launches **tvdsvr**, it waits 30 seconds for **tvdsvr** to respond with a successful connection message. If the connection is not made in this time, TotalView times out. You can enter a value from 1 to 3600 seconds (1 hour).

In addition, you can preset the timeout value using an X resource. See "TOTALVIEW*SERVERLAUNCHTIMEOUT" on page 290 for more information.

If you notice that TotalView fails to launch **tvdsvr** (as shown in the **xterm** window from which you started the debugger) before the timeout expires, pressing Ctrl-C in any TotalView window aborts the launch request.

If you make a mistake or decide you want to go back to TotalView's default settings, select the **Defaults** button. This command also overrides changes you made using an X resource. TotalView does not immediately change settings after you press the **Defaults** button; instead, it waits until you select the **OK** button.

## Bulk Launch Window Options

The **Bulk Launch Window** dialog box lets you change the bulk launch command, disable bulk launch, and alter connection timeouts used by TotalView when it launches the **tvdsvr** programs.

The Root Window's popup menu contains the **Bulk Launch Window** command. After selecting this command on an SGI MIPS machine, TotalView displays the dialog box shown in Figure 28:

If you are running on an RS/6000 AIX machine, the defaults are different.

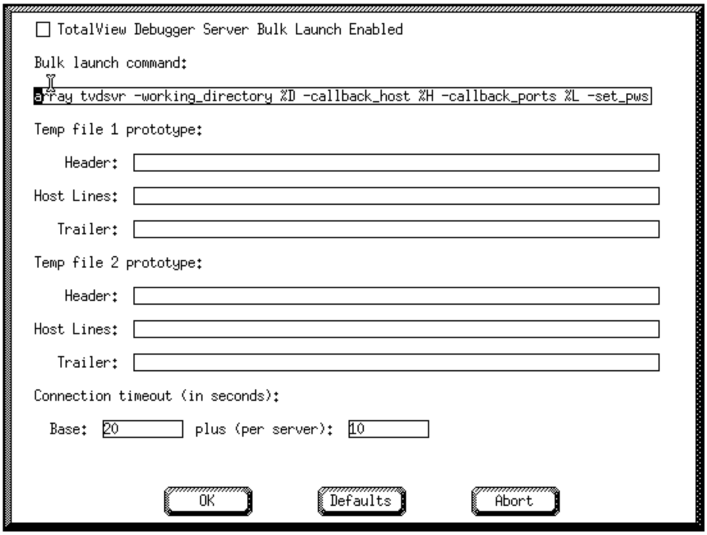



FIGURE 28: **Bulk Launch Window**

**TotalView Debugger Server Bulk Launch Enabled**

If the checkbox is selected, TotalView will bulk launch the TotalView Debugger Server (**tvdsvr**). By default, bulk launch is disabled.

**Bulk launch command**

If bulk launch is enabled, TotalView will use this command to launch **tvdsvr**. For information on this command and its options, see "*Bulk Server Launch on an SGI MIPs Machine*" on page 61 and "*Bulk Server Launch on an IBM RS/6000 AIX Machine*" on page 62.

**Temp file prototypes**

These fields can be used to create temporary files in bulk launch operations. For information on these fields, see Chapter 13 "*TotalView Debugger Server Command Syntax*" on page 311.

**Connection timeout**

After TotalView launches **tvdsvr** processes, it waits 20 seconds plus 10 seconds for each server launched for them to respond with successful connection messages. (The text boxes let you change these values.) If the connections are not made in this time, TotalView times out.

The **Base** timeout value can be from 1 to 3600 seconds (1 hour). The incremental (**plus**) value is from 1 to 360 seconds. You can preset these timeout values using X resources. See "TOTALVIEW*BULKLAUNCHBASETIMEOUT" on page 278 and "TOTALVIEW*BULKLAUNCHINCRTIMEOUT" on page 278 for more information.

If you notice that TotalView fails to launch the **tvdsvr** processes (as shown in the **xterm** window from which you started TotalView) before the timeout expires, pressing Ctrl-C in any TotalView window aborts the launch request.

## Starting the Debugger Server Manually

If you cannot tailor the auto-launch feature to work on your system, you can start the debugger server manually. The major disadvantage of this method is that it is not secure: other users could connect to your instance of **tvdsvr** and begin using your UNIX UID.

Here is how you manually start **tvdsvr**:

1. From the Root Window, select the **Server Launch Window** command. The dialog box shown in Figure 27 appears.
2. Deselect the **TotalView Debugger Server Auto Launch Enabled** checkbox to disable the auto-launch feature and then select **OK**.
3. Log in to the remote machine and start **tvdsvr**:

   **tvdsvr –server**

   If you do not (or cannot) use the default port number (4142), you will need to use the **–port** or **–search_port** options For details, refer to "*TotalView Debugger Server Command Syntax*" on page 311.

   After printing out the port number and the assigned password, the server begins listening for connections. Be sure to make note of the password; you will need to enter it later in step 5.

   NOTE **Because using the -server option is not secure, it must be explicitly enabled. For details, see "*–server*" on page 314.**

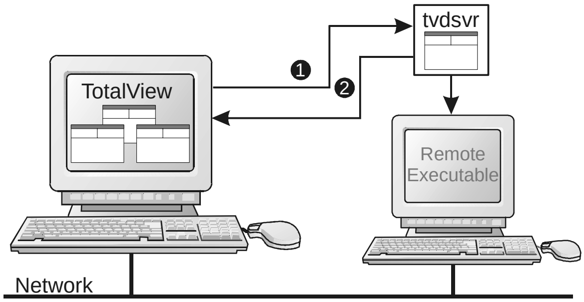
**4** From the Root Window, select the **New Program Window** command. Enter the name in the **Executable file name** field of the dialog that appears and the *hostname*:*portnumber* in the **Program location** field.

Select **OK**.

**5** TotalView now tries to connect to **tvdsvr**.

When TotalView prompts you for the password, enter the password that **tvdsvr** displayed in step 3.

Figure 29 summarizes the steps used when you start **tvdsvr** manually.




❶ Makes connection
❷ Listens

FIGURE 29: **Manual Launching of Debugger Server**

## Single Process Server Launch Command

By default, TotalView uses the following command string when it automatically launches the debugger server for a single process:

```
%C %R –n "tvdsvr –working_directory %D –callback %L \
                –set_pw %P –verbosity %V"
```

where:

| | |
|---|---|
| %C | Expands to the name of the server launch command being used. On most platforms, this is **rsh**. On HP, this command is **remsh**. If the **TVDSVRLAUNCHCMD** environment variable exists, TotalView will use its value instead of its platform-specific default value. |
| %R | Expands to the host name of the remote machine that you specified in the **New Program Window** command. |

**–n** Tells the remote shell to read standard input from **/dev/null**.

**–working_directory %D**
Makes **%D** the directory to which TotalView will be connected. **%D** expands to the absolute pathname of the directory.

Using this option assumes that the host machine and the target machine mount identical filesystems. That is, the pathname of the directory to which TotalView is connected must be identical on the host and target machines.

After performing this operation, the shell will start the TotalView Debugger Server using the **tvdsvr** command.

You must make sure that TotalView directory is on your path on the remote machine.

**–callback %L** Establishes a connection from **tvdsvr** to TotalView using the indicated host name and port number. **%L** expands to the host name and TCP/IP port number (*hostname:port*) on which TotalView is listening for connections from **tvdsvr**.

**–set_pw %P** Sets a 64-bit password. TotalView must supply this password when **tvdsvr** establishes the connection with it. **%P** expands to the password that TotalView automatically generated. For more information on this password, see Chapter 13 "*TotalView Debugger Server Command Syntax*" on page 311.

**–verbosity %V** Sets the verbosity level of the TotalView Debugger Server. **%V** expands to the current TotalView verbosity setting.

You can also use the **%H** option with this command. This option is discussed in "*Bulk Server Launch on an* SGI MIPs *Machine*" on page 61.

To change the server launch command each time you start TotalView, use the X Resource "TOTALVIEW*SERVERLAUNCHSTRING" on page 289.

For information on the complete syntax of the **tvdsvr** command, refer to "*TotalView Debugger Server Command Syntax*" on page 311.

## Bulk Server Launch on an SGI MIPs Machine

On an SGI machine, the launch string used for a bulk server launch is similar to the single process server launch and is:

```
array tvdsvr –working_directory %D –callback_host %H \
         –callback_ports %L –set_pws %P –verbosity %V
```

where:

**–working_directory %D**
: Makes **%D** the directory to which TotalView will be connected. **%D** expands to the absolute pathname of the directory.

  Note that the command assumes that the host machine and the target machine mount identical filesystems. That is, the pathname of the directory to which TotalView is connected must be identical on both the host and target machines.

  After performing this operation, the TotalView Debugger Server is started.

**–callback_host %H**
: Names the host upon which the callback is made. **%H** expands to the hostname of the machine upon which TotalView is running.

**–callback_ports %L**
: Names the ports on the host machines that are used for callbacks. **%L** expands to a comma-separated list of the host names and TCP/IP port numbers (*hostname*:*port*,*hostname*:*port*…) on which TotalView is listening for connections from **tvdsvr**.

**–set_pws %P**
: Sets 64-bit passwords. TotalView must supply these passwords when **tvdsvr** establishes the connection with it. **%P** expands to a comma-separated list of 64-bit passwords that TotalView automatically generates. For more information, see Chapter 13 "*TotalView Debugger Server Command Syntax*" on page 311.

**–verbosity %V**
: Sets the verbosity level of the TotalView Debugger Server. **%V** expands to the current TotalView verbosity setting.

In some circumstances, you may need to add the **%S** substitution character to your command string. This expands a comma-separated list of the port numbers that the server should use when it makes a callback to TotalView.

You must enable **tvdsvr**'s use of the array command by adding the following information to the **/usr/lib/array/arrayd.conf** file:

```
#
# Command that allow invocation of the TotalView Debugger
# server when performing a Bulk Server Launch.
#
command tvdsvr
        invoke /opt/totalview/bin/tvdsvr %ALLARGS
        user %USER
        group %GROUP
        project %PROJECT
```

For information on the complete syntax of the **tvdsvr** command, refer to "*TotalView Debugger Server Command Syntax*" on page 311.

## Bulk Server Launch on an IBM RS/6000 AIX Machine

On an IBM RS/6000 AIX machine, the launch string used for a bulk server launch is:

```
%C %H "poe –pgmmodel mpmd –resd no –tasks_per_node 1
                –procs %N –hostfile %t1 –cmdfile %t2"
```

where the elements unique to TotalView are:

%N — Expands to the number of servers that will be launched.

%t1 — A temporary file created by TotalView that contains a list of the hosts upon which **tvdsvr** will run.

TotalView generates this information by expanding the **%R** symbol in the **Bulk Launch** Window.

%t2 — A file that contains the commands to start the **tvdsvr** processes on each machine. TotalView creates these lines by expanding the following template:

```
tvdsvr –working_directory %D \
    –callback %L –set_pw %P –verbosity %V
```

## Disabling Auto-Launch

If after changing the auto-launch options, TotalView still cannot automatically start **tvdsvr**, you must disable the auto-launch and start **tvdsvr** manually. Here are three ways for doing this:

- If you change the auto-launch options (see "*Single Process Server Launch Options*" on page 55), you must also deselect the **TotalView Debugger Server Auto Launch Enabled** checkbox in the **Server Launch Window** dialog box. This disables auto-launch for the current TotalView session.
- When you debug the remote process, as described in "*Debugging Remote Processes*" on page 51, enter a host name and port number in the bottom section of the **New Program Window** dialog box. This disables auto-launch for the current connection.
- Set an X resource that disables auto-launch. For more information, refer to "TOTALVIEW*SERVERLAUNCHENABLED" on page 289. This resource disables auto-launch for all TotalView session.

NOTE **If you disable the auto-launch feature, you must start tvdsvr *before* you load a remote executable or attach to a remote process.**

## Changing the Remote Shell Command

Some environments require that you create your own auto-launch command. You might do this, for example, if your remote shell command does not provide the security required by your site and you need to invoke remote processes using a more secure command.

If you create your own auto-launch command, you must invoke **tvdsvr** using the **–callback** and **–set_pw** arguments.

If you are not sure whether **rsh** (or **remsh** on HP machines) works at your site, try typing "**rsh** *hostname*" (or "**remsh** *hostname*") from an **xterm**, where *hostname* is the name of the host upon which you want to invoke the remote process. If this command prompts you for a password, you must add the host name of the host machine to your **.rhosts** file on the target machine.

For example, you could use a combination of the **echo** and **telnet** commands:

```
echo %D %L %P %V; telnet %R
```

Once **telnet** establishes a connection to the remote host, you could use the **cd** and **tvdsvr** commands directly, using the values of **%D**, **%L**, **%P,** and **%V** that were displayed by the **echo** command. For example:

```
cd directory
tvdsvr –callback hostname:portnumber –set_pw password
```

If your machine does not have a command for invoking a remote process, you cannot use the auto-launch feature and should disable it.

For information on the **rsh** and **remsh** commands, refer to the manual page supplied with your operating system.

## Changing the Arguments

You can also change the command-line arguments passed to **rsh** (or whatever command you use to invoke the remote process).

For example, if the host machine does not mount the same filesystems as your target machine, the debugger server may need to use a different path to access the executable being debugged. If this is the case, you could change **%D** to the directory used on the target machine.

If the remote executable reads from standard input, you cannot use the **–n** option with your remote shell command because this option causes the remote executable to receive an EOF immediately on standard input. If you omit **–n**, the remote executable reads standard input from the **xterm** in which you started TotalView. This means that you should invoke **tvdsvr** from another **xterm** window if your remote program reads from standard input. Here's an example:

```
%C %R "cd %D && xterm –display hostname:0 –e tvdsvr \
        –callback %L –set_pw %P –verbosity %V"
```

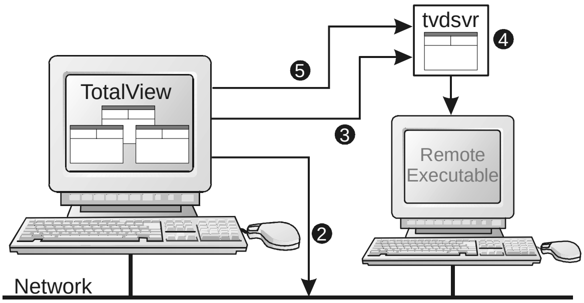
Now, each time TotalView launches **tvdsvr**, a new **xterm** appears on your screen to handle standard input and output for the remote program.

## Auto-launch Sequence

If you want to know more about auto-launch, here is the sequence of actions carried out by you, TotalView, and **tvdsvr**:

1 With the **New Program Window** command, you specify the host name of the machine on which you want to debug a remote process, as described in "*Debugging Remote Processes*" on page 51.
2 TotalView begins listening for incoming connections.
3 TotalView launches the **tvdsvr** process with the server launch command. ("*Single Process Server Launch Command*" on page 59 describes this command.)
4 The **tvdsvr** process starts on the remote machine.
5 The **tvdsvr** process establishes a connection with TotalView.

Figure 30 summarizes these actions.




❷ Listens
❸ Invokes commands
❹ tvdsvr starts
❺ Makes connection

FIGURE 30: **Root Window Showing Process and Thread Status**

# Debugging Over a Serial Line

TotalView allows you to debug over a serial line as well as TCP/IP sockets. However, if a network connection exists, you will probably want to use TCP/IP sockets remote debugging for better performance.

You will need to have two connections to the target machine. One connection is for the console and the other is for TotalView's use. Do not try to use one serial line; TotalView cannot share a serial line with the console.

Figure 31 illustrates a TotalView debugging session using a serial line. In this example, TotalView is communicating over a dedicated serial line with a TotalView Debugger Server running on the target host. A VT100 terminal is connected to the target host's console line, allowing you to type commands on the target host.

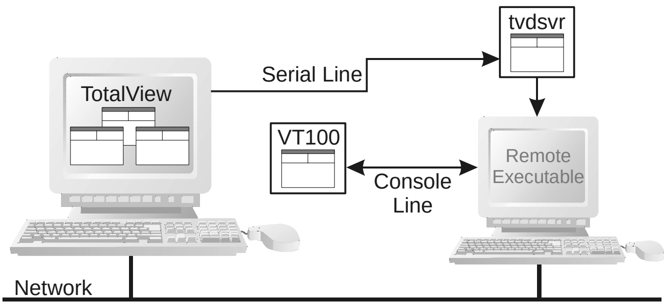



FIGURE 31: **TotalView Debugging Session Over a Serial Line**

## Start the TotalView Debugger Server

To start a TotalView debugging session over a serial line from the command line, you must first start the TotalView Debugger Server (**tvdsvr**).

Using the console connected to the target machine, start **tvdsvr** and specify the name of the serial port device on the target machine. The syntax of this command is:

**tvdsvr –serial** *device*[**:baud=***num*]

where:

| | |
|---|---|
| *device* | The name of the serial line device |
| *num* | The serial line's baud rate; if you omit the baud rate, TotalView uses a default value of **38400** |

For example:

```
tvdsvr –serial /dev/com1:baud=38400
```

After it is starts, the TotalView Debugger Server will wait for TotalView to establish a connection.

## Starting TotalView on a Serial Line

Start TotalView on the host machine and include the name of the serial line device. The syntax of this command is:

**totalview –serial** *device*[**:baud=***num*] *filename*

where:

| | |
|---|---|
| *device* | The name of the serial line device on the host machine |
| *num* | The serial line's baud rate; if you omit the baud rate, TotalView uses a default value of **38400** |
| *filename* | The name of the executable file |

For example:

```
totalview –serial /dev/term/a test_pthreads
```

## New Program Window

Here is the procedure for starting a TotalView debugging session over a serial line when you are already in TotalView:

**1** Start the TotalView Debugger Server. (This is discussed in "*Start the TotalView Debugger Server*" on page 66).

**2** Issue the **New Program Window** command from the Root Window to display the **New Program Window** dialog box, shown in Figure 32.

Enter the name of the executable file in the **Executable file name** field.

Enter the name of the serial line device in the **Program location** field, and select the **Serial line** radio button.

**3** Select **OK**.

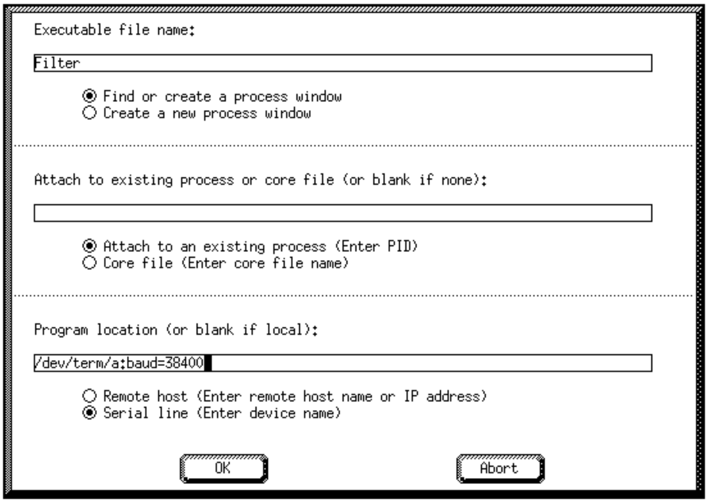



FIGURE 32: **New Program Window Dialog Box**

Chapter 5

# Setting Up Parallel Debugging Sessions

This chapter explains how to set up TotalView parallel debugging sessions for applications that use the following parallel execution models:

- MPI (and MPICH)
- OpenMP
- ORNL PVM and Compaq DPVM
- SGI "shared memory" (shmem)
- Portland Group HPF

## Debugging MPI Applications

You can use TotalView to debug your Message Passing Interface (MPI) programs. With TotalView, you can:

- Automatically acquire processes at start-up.
- Attach to a parallel program and automatically acquire the parallel processes.
- Display the message queue state of a process.

Automatic process acquisition at start-up is supported for the following MPI implementations:

- MPICH version 1.1.0 or later running on any platform that is supported by both TotalView and MPICH (see "*Debugging MPICH Applications*" on page 70). (You are strongly urged to use a later version of MPICH. Information on versions that work with TotalView can be found in the TOTALVIEW RELEASE NOTES.)

- Compaq MPI (DMPI) running on Compaq Alpha (see "*Debugging Compaq MPI Applications*" on page 74).
- HP MPI running on HP PA-RISC 1.1 or 2.0 processors (see "*Debugging HP MPI Applications*" on page 75).
- IBM MPI Parallel Environment (PE) running on AIX on RS/6000 and SP (see "*Debugging IBM MPI (PE) Applications*" on page 76).
- SGI MPI running on IRIX on MIPS processors (see "*Debugging SGI MPI Applications*" on page 80).
- QSW RMS2 running on Compaq AlphaServer SC systems (see "*Debugging QSW RMS2 Applications*" on page 81).

For more information on message queue display, see "*Displaying Message Queue State*" on page 82.

For tips on debugging parallel applications, see "*Parallel Debugging Tips*" on page 110.

# Debugging MPICH Applications

To debug Message Passing Interface/Chameleon Standard (MPICH) applications you must use MPICH version 1.1.0 or later on a homogenous collection of machines. If you need a copy of MPICH, you can obtain it at no cost from Argonne National Laboratory at **www.mcs.anl.gov/mpi**. (You are strongly urged to use a later version of MPICH. Information on versions that work with TotalView can be found in the TOTALVIEW RELEASE NOTES.)

You should configure the MPICH library to use either the **ch_p4**, **ch_shmem**, **ch_lfshmem**, or **ch_mpl** devices. For networks of workstations, **ch_p4** is the normal default. For shared-memory SMP machines, use **ch_shmem**. On an IBM SP machine, use the **ch_mpl** device. The MPICH source distribution includes all of these devices and you can choose one when you configure and build MPICH.

**NOTE** When configuring MPICH, you must ensure that the MPICH library maintains all of the information required by TotalView. Use the –debug option with the MPICH configure command. In addition, the TotalView Release Notes contains information on patching your MPICH 1.1.0 distribution.

## Starting TotalView on an MPICH Job

You must have both TotalView (**totalview**) and the TotalView Debugger Server (**tvdsvr**) in your path when you start an MPICH job under TotalView's control. Use the MPICH **mpirun** command that you customarily use and add the **–tv** option:

**mpirun** [ MPICH-*arguments* ] **–tv** *program* [ *program-arguments* ]

For example:

```
mpirun –np 4 –tv sendrecv
```

The MPICH **mpirun** command extracts the value of the **TOTALVIEW** environment variable and then uses its value when it starts the first process in the parallel job. Therefore, setting this environment variable lets you use a different TotalView, or pass command line options to TotalView.

For example, here's the C shell command that tells **mpirun** to start the TotalView debugger using the **–no_stop_all** option:

```
setenv TOTALVIEW "totalview –no_stop_all"
```

On workstations, TotalView starts the first process of your job, the master process, under its control. Then, you can set breakpoints, and debug your code as usual.

On the IBM SP machine, the **mpirun** command uses the **poe** command to start an MPI job. While you still must use the MPICH **mpirun** (and it **-tv** option) command to start an MPICH job, the way you start MPICH differs since you are using **poe**. For details of using TotalView with **poe**, see "*Starting TotalView on a PE Job*" on page 77.

When you let code run through the call to **MPI_Init()**, TotalView automatically acquires the other processes that make up your parallel job. A dialog box appears that asks if you want to stop the spawned processes. This allows you to stop all of the processes in **MPI_Init()** so you can check their states before they run too far, as shown in Figure 33.

TotalView automatically copies breakpoints from the master process to the slave processes as it acquires them. You do not have to first stop the slave

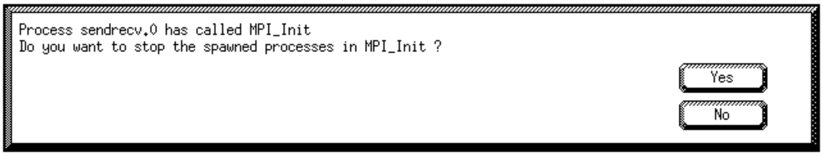



FIGURE 33: **Stopping Spawned Processes Dialog Box**

processes in **MPI_Init()**. Next, TotalView updates the Root Window to show all the newly acquired processes.

## Attaching to an MPICH Job

TotalView allows you to attach to an MPICH application even if it was not started under TotalView's control. Here is how you do this:

**1** Start TotalView in the normal manner.

**2** Issue the **Show All Unattached Processes** command from the Root Window. A window appears on your screen displaying the **Processes that TotalView doesn't own** window, as shown in the following figure.

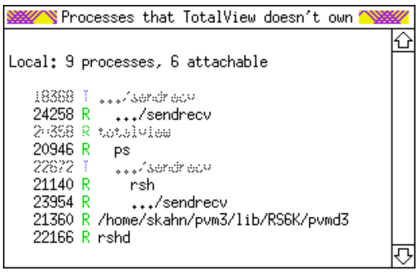



FIGURE 34: **Processes that TotalView doesn't own Window**

**3** On workstation clusters, attach to the first MPICH process by diving into it.

Normally, the first MPICH process is the highest process with the correct image name in the process list. Other instances of the same executable can be:

- The **p4** listener processes if you have configured MPICH with **ch_p4**.
- Additional slave processes if you have configured MPICH with **ch_shmem** or **ch_lfshmem**.

- ➤ Additional slave processes if you have configured MPICH with **ch_p4** and have a machine file that places multiple processes on the same machine.
- ➤ On an IBM SP, attach to the **poe** process that started your job. For details, see "*Starting TotalView on a* PE *Job*" on page 77.

4 After you attach to the processes, TotalView asks if you also wish to attach to the slave MPICH processes. If you do, press **Return** or choose **Yes**. If you do not, select **No**.

If you choose **Yes**, TotalView starts the server processes and acquires all of the MPICH processes.

In some situations, the processes you expect to see may not exist (for example, they may have crashed or exited). TotalView acquires all the processes it can and then warns you if it could not attach to some of them. If you attempt to dive into a process that no longer exists (for example, through the source or target fields of a message state display), TotalView tells you that the process no longer exists.

## MPICH P4 procgroup Files

If you are using MPICH with a P4 **procgroup** file (by using the **–p4pg** option), make sure you use the *same* absolute path name in your **procgroup** file and on the **mpirun** command line. If your **procgroup** file contains different path names that resolve to the same executable, TotalView treats each path name as a separate instance of the executable, which causes debugging problems.

You must use the *same* absolute pathname for the executable on the TotalView debugger's command line and in the **procgroup** file. For example:

```
% cat p4group
local 1 /users/smith/mympichexe
bigiron 2 /users/smith/mympichexe
% mpirun –p4pg p4group –tv /users/smith/mympichexe
```

In this example, TotalView:

1 Reads the symbols from the executable **mympichexe** only once.

2 Places MPICH processes in the same TotalView share group.

**3** Names the processes **mypichexe.0**, **mympichexe.1**, **mympichexe.2**, and **mympichexe.3**.

If Totalview assigns names such as **mympichexe<mympichexe>.0**, a problem occurred and you should check the contents of your **procgroup** file and **mpirun** command line.

# Debugging Compaq MPI Applications

You can debug Compaq MPI applications on the Compaq Alpha platform. To use TotalView with Compaq MPI, you must use Compaq MPI version 1.7 or later.

## Starting TotalView on a Compaq MPI Job

Compaq MPI programs are normally started using the **dmpirun** command. You would use a very similar command to start an MPI program under the debugger's control:

**totalview dmpirun –a** *dmpirun-command-line*

This invokes TotalView and tells it to show you the code for the main program in **dmpirun**. Since you are not usually interested in debugging this code, you should let the program run by using the **Go Process** command.

The **dmpirun** command runs and starts all of the MPI processes. TotalView acquires them and then ask if you want to stop them all.

**NOTE** Problems re-running Compaq MPI programs under TotalView control due to resource allocation issues within Compaq MPI can occur. Consult the Compaq MPI manuals and release notes for information on cleaning up the MPI system state using mpiclean.

## Attaching to a Compaq MPI Job

To attach to a running Compaq MPI job, attach to the **dmpirun** process that started the job. The procedure for attaching to a **dmpirun** process is the same as the procedure for attaching to other processes. For details, see "*Attaching to Processes*" on page 33.

Once you have attached to the **dmpirun** process, TotalView displays the same dialogue as it does with MPICH. (See step 4 on page 73, included in "*Attaching to an* MPICH *Job*" on page 72.)

# Debugging HP MPI Applications

You can debug HP MPI applications on the a PA-RISC 1.1 or 2.0 processor. To use TotalView with HP MPI, you must use HP MPI version 1.6.

## Starting Totalview on an HP MPI Job

TotalView lets you start an MPI program in three ways:

**totalview** *program* **–a** *mpi-arguments*

This command tells TotalView to start the MPI process. TotalView will then show you the machine code for the HP MPI **mpirun** executable. Since you are not usually interested in debugging this code, you should let the program run by using the **Go Process** command.

**mpirun** *mpi-arguments* **–tv** *program*

This command tells MPI that it should start TotalView.

**mpirun** *mpi-arguments* **–tv -f** *startup_file*

This third method tells MPI that it should start TotalView and then start the MPI processes as they are defined within the *startup_file* script. This script names the processes that will be started. Typically, this file has contents that are similar to:

```
-h localhost -np 1 sendrecv
-h localhost -np 1 sendrecva
```

In this example, **sendrecv** and **sendrecva** are two different exeutable programs. (Your HP MPI documentation describes the contents of this file.)

Just before **mpirun** starts the MPI processes, TotalView acquires them and asks if you want to stop the process before it starts executing. If your answer is **yes**, TotalView halts them before they enter the main program. You can then enter breakpoints.

### Attaching to an HP MPI Job

To attach to a running HP MPI job, attach to the HP MPI **mpirun** process that started the job. The procedure for attaching to a **mpirun** process is the same as the procedure for attaching to any other process. For details, see "*Attaching to Processes*" on page 33.

Once you have attached to the HP MPI **mpirun** process, TotalView displays the same dialog as it does with MPICH. (See step 4 on page 73 of "*Attaching to an MPICH Job*" on page 72.)

## Debugging IBM MPI (PE) Applications

You can debug IBM MPI Parallel Environment (PE) applications on the IBM RS/6000 and SP platforms.

To take advantage of TotalView's automatic process acquisition capabilities, you must be running release 2.2 or later of the Parallel Environment for AIX. If you aren't running release 2.2, you can run TotalView on release 2.1 if you also load PTF 15.

See "*Displaying Message Queue State*" on page 82 for message queue display information.

### Preparing to Debug a PE Application

The following sections describe steps you must perform before you can display a PE application.

**Switch-Based Communication:** If you are using switch-based communications (either "IP over the switch" or "user space") on an SP machine, you must configure your PE debugging session so that TotalView can use "IP over the switch" for communicating with the TotalView Debugger Server, by setting **adaptor_use** to **shared** and **cpu_use** to **multiple**, as follows:

- If you are using a PE host file, add **shared multiple** after all host names or pool IDs in the host file.

- Whether or not you have a PE host file, enter the following arguments on the **poe** command line:

  **–adaptor_use shared –cpu_use multiple**

  If you do not want to set these arguments in the **poe** command line, set the following environment variables before starting **poe**:

  **setenv MP_ADAPTOR_USE shared**
  **setenv MP_CPU_USE multiple**

  When using "IP over the switch," the default is usually **shared adapter use** and **multiple cpu use**; to be safe, set it explicitly using one of these techniques.

When you are using switch-based communications, you must run TotalView on one of the SP or SP2 nodes. Since TotalView uses "IP over the switch" in this case, you cannot run TotalView on an RS/6000 workstation.

**Remote Login:** You must to be able to use remote login using **rsh.** To do this, add the host name of the remote node to the **/etc/hosts.equiv** file or to your **.rhosts** file.

When the program is using switch-based communications, TotalView tries to start the TotalView Debugger Server using the **rsh** command with the switch host name of the node.

**Timeout:** TotalView automatically sets the **timeout** value at 600 seconds. If you get communications timeouts, you may need to set the value at a higher number, as in the following example:

```
setenv MP_TIMEOUT 1200
```

**NOTE** **The timeout value cannot be set using the poe command line.**

## Starting TotalView on a PE Job

Parallel Environment (PE) programs are run from the command line using the following syntax:

*program* [ *arguments* ] [ PE_*arguments* ]

They can also be run the **poe** command:

**poe** *program* [ *arguments* ] [ PE_*arguments* ]

However, if you start TotalView on a PE application, you must start use the **poe** executable as TotalView's target. The syntax of the command is:

**totalview poe –a** *program* [ *arguments* ] [ PE_*arguments* ]

For example:

```
totalview poe –a sendrecv 500 –rmpool 1
```

## Setting Breakpoints

After TotalView is running, you can start the **poe** process; this process then starts the parallel processes. Issue the **Go Process** command from the Process Window. TotalView displays a dialog box that asks if you want to stop the parallel tasks, as shown in the following figure.

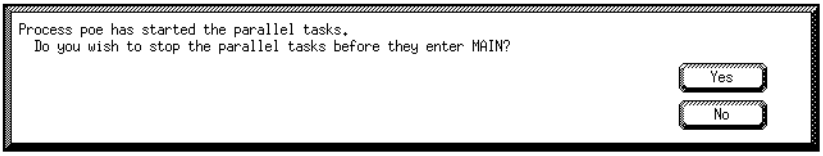



FIGURE 35: **Parallel Tasks Dialog Box**

If you want to set breakpoints in your code at this point, answer **Yes** to stop the processes. TotalView initially stops the parallel tasks, which also allows you to set breakpoints. After a program window for the first parallel task appears, you can set breakpoints and control the parallel tasks using normal TotalView commands.

If you have already set and saved breakpoints in a file and want to reload the file, answer **No**. After TotalView reloads your breakpoints, the parallel tasks continue running.

## Starting Parallel Tasks

After you set breakpoints, you can start all of the parallel tasks by issuing the **Go Group** command from the Parallel Task Program Window.

**NOTE** No parallel tasks will reach the first line of code in **main** until all parallel tasks start.

You should be very cautious in placing breakpoints at or before the line that contains the call to **MPI_Init()** (or **MPL_Init()**) because timeouts can occur during the initialization process. Once you allow any of the parallel processes to proceed into the **MPI_Init()** or **MPL_Init()** call, you should allow all of the parallel processes to proceed through this call within a short time. For more information on this, see "*Avoid unwanted timeouts*" on page 113.

## Attaching to a PE Job

To take full advantage of TotalView's **poe**-specific automation, you need to attach to **poe** itself, and let TotalView automatically acquire the **poe** processes on its various nodes. This set of acquired processes will include the processes you want to debug.

You attach to the **poe** processes the same way you attach to other processes. For details, see "*Attaching to Processes*" on page 33.

### Attaching From a Node Running poe

To attach TotalView to **poe** from the node running **poe**, start TotalView in the directory of the debug target. If you cannot start TotalView in the debug target directory, you can start TotalView by editing the TotalView Debugger Server (**tvdsvr**) command line before attaching to **poe**. See "*Single Process Server Launch Command*" on page 59.

In the TotalView Root Window, bring up the **Unattached Processes** Window, find the **poe** process list in your Root Window, and attach to it by diving into it. TotalView launches TotalView Debugger Servers as necessary.

TotalView updates the Root Window and opens a Process Window for the **poe** process, which you just dove on. In the Root Window, find the process you want to debug and dive on it to open a Process Window from which you can control and debug the target process.

If source code files are available but are not displayed in the Source Code Pane, you probably have not told TotalView where these files reside. You can tell TotalView where the files are by invoking the **Display/Directory/Edit** command.

### Attach From a Node Not Running poe

To attach TotalView to **poe** from a node not running **poe**, follow the same procedures as in attaching from a node running **poe**, except, since you did not run TotalView from the node running **poe** (the start-up node), you will not be able to see **poe** on the process list in your Root Window and you will not be able to start it by diving into it.

To get **poe** on the process list in your Root Window, connect TotalView to the start-up node. For details, see "*Connecting to Remote Machines*" on page 53 and "*Attaching to Processes*" on page 33. Then, update the list of processes in the **Processes that TotalView doesn't own** window by selecting **Update Process List** from the menu. In the *startup_node_name* area, look for the process named **poe** and continue as if attaching from a node running **poe**.

# Debugging SGI MPI Applications

TotalView can acquire processes started by SGI MPI version 3.1, which is part of the Message Passing Toolkit (MPT) 1.2 package.

Message queue display is supported by release 1.3 of the Message Passing Toolkit. See "*Displaying Message Queue State*" on page 82 for message queue display.

## Starting Totalview on a SGI MPI Job

SGI MPI programs are normally started using the **mpirun** command. You would use a very similar command to start an MPI program under the TotalView debugger's control:

**totalview mpirun –a** *mpirun-command-line*

This invokes TotalView and tells it to show you the machine code for SGI MPI **mpirun**. Since you are not usually interested in debugging this code, you should let the program run by using the **Go Process** command.

The SGI MPI **mpirun** command runs and starts all MPI processes. After TotalView acquires them, it asks if you want to stop them at start-up. If you answer *yes*, TotalView halts them before they enter the main program. You can then enter breakpoints.

If you set a verbosity level that allows informational messages, TotalView also prints a message showing the name of the array and the value of the array services handle (**ash**) to which it is attaching.

## Attaching to an SGI MPI Job

To attach to a running SGI MPI job, attach to the SGI MPI **mpirun** process that started the job. The procedure for attaching to a **mpirun** process is the same as the procedure for attaching to any other process. For details, see "*Attaching to Processes*" on page 33.

Once you have attached to the SGI MPI **mpirun** process, TotalView displays the same dialog as it does with MPICH. (See step 4 on page 73 of "*Attaching to an MPICH Job*" on page 72.)

# Debugging QSW RMS2 Applications

TotalView supports automatic process acquisition on AlphaServer SC systems that use Quadrics' RMS2 resource management system with the QSW switch technology.

Message queue display for RMS2 is not yet supported by the system.

## Starting TotalView on an RMS2 Job

To start a parallel job under the control of TotalView, use TotalView as though you were debugging the **prun** command itself:

**totalview prun –a** *prun-command-line*

TotalView starts up and shows you the machine code for RMS2 **prun**. Since you are not usually interested in debugging this information, you should let the program continue running by using the **Go Process** command.

The RMS2 **prun** command executes and starts all of the MPI processes. TotalView acquires them and then asks if you want to stop them at start-up. If you do stop them, TotalView halts them before they enter the main program. You can then enter breakpoints.

### Attaching to an RMS2 Job

To attach to a running RMS2 job, attach to the RMS2 **prun** process that started the job.

You attach to the **prun** processes the same way you attach to other processes. For details on attaching to processes, see "*Attaching to Processes*" on page 33.

Once you have attached to the RMS2 **prun** process, TotalView displays the dialog as it does with MPICH. (See step 4 on page 73 of "*Attaching to an MPICH Job*".)

## Displaying Message Queue State

The TotalView message queue display (MQD) allows you to display the message queue state of your MPI program. This lets you determine the cause of message passing deadlocks.

To use the message queue display feature, you must be using the following versions of MPI, as follows:

- MPICH version 1.1.0 or later.
- Compaq Alpha MPI (DMPI) version 1.7.
- HP HP-UX version 1.6.
- IBM MPI Parallel Environment (PE) version 2.3 or 2.4; but only for programs using the threaded IBM MPI libraries. MQD is not available with earlier releases, or with the non-thread-safe version of the IBM MPI library. Therefore, to use TotalView MQD with IBM MPI applications, you must compile and link your code using the **mpcc_r**, **mpxlf_r**, or **mpxlf90_r** compilers.
- For SGI MPI TotalView message queue display, you must obtain the Message Passing Toolkit (MPT) release 1.3 or later. Check with SGI for availability.

## Message Queue Display Basics

After an MPI process returns from the call to **MPI_Init()**, you can display the internal state of the MPI library by issuing the **Message State Window** command in the **Process State Info** menu of the Process Window. TotalView opens a Message State Window for the process, as shown in Figure 36.

The **Message State** Window displays the state of each of the MPI communicators that exist in the process. In some MPI implementations, such as MPICH, user-visible communicators are implemented as two internal communicator structures, one for point-to-point and the other for collective operations. TotalView shows both structures.

**NOTE** You cannot edit any of the fields in the Message State Window.

The contents of the Message State Window are only valid when the process is stopped. (See Figure 36.)

For each communicator, TotalView displays the following fields:

- **Communicator Name.** MPI names the pre-defined communicators such as **MPI_COMM_WORLD()**. MPICH 1.1 and Compaq MPI also provide the MPI-2 **MPI_NAME_PUT()** and **MPI_NAME_GET()** communicator naming functions, so you can associate a name with a communicator. If you use **MPI_NAME_PUT()** to name a communicator, TotalView uses the name you gave it when displaying the communicator, so you do not have to guess which communicator is which.

  IBM MPI and SGI MPI do not implement the MPI-2 communicator naming functions; this means that only pre-defined communicators are named. For user-created communicators, the integer value that represents the communicator is displayed. This is the value that a variable of type **MPI_Communicator** has if it represents the given communicator.
- **Comm_size** is the number of processes in the communicator. This is the same as the result of **MPI_Comm_size()** applied to the communicator.
- **Comm_rank** is the rank in the communicator of the process that owns the Message State Window. This is the same result that you would get if you had applyed **MPI_Comm_rank** to the communicator in this process.
- List of pending unexpected messages; that is, messages that were sent to this communicator but have not yet matched with a receive.

```
Message State for "sendrecv.0" (20003.1)
MPI_COMM_WORLD
Comm_size                2
Comm_rank                0
Pending receives    : none
Unexpected messages : none
Pending sends       : none
MPI_COMM_WORLD_collective
Comm_size                2
Comm_rank                0
Pending receives    : none
Unexpected messages : none
Pending sends       : none
MPI_COMM_SELF
Comm_size                1
Comm_rank                0
Pending receives    : none
Unexpected messages : none
Pending sends       : none
MPI_COMM_SELF_collective
Comm_size                1
Comm_rank                0
Pending receives    : none
Unexpected messages : none
Pending sends       : none
```

FIGURE 36: **Message State Window**

- List of pending receive operations.
- List of pending send operations.

## Message Operations

For each communicator, TotalView displays a list of pending receive operations, pending unexpected messages, and pending send operations. Each operation has an index value displayed in square brackets ([*n*]), and each operation can include the following fields:

**Actual Source** If the **Status** is **Complete** and the **Source** is **ANY**, the receiving process.

**Actual Tag** If the **Status** is **Complete** and the **Tag** value is **ANY**, this is the received tag value.

**Buffer Length or Received Length**
The buffer length in bytes, shown in decimal and hexadecimal.

**Function** The MPI function (IBM MPI only). The name of the MPI function associated with the operation; for example, **MPI_Irecv()**.

**Source or Target**
The source or target process. **Source** is the process from which the message should be received. **Target** is the process to which the message is being sent. This field shows the index of the process in the communicator, and the process name in parentheses. The display shows **ANY** if the message is being received from **MPI_ANY_SOURCE**.

Dive into this field to display a Process Window.

**Status** The status of the operation. Operation status can be **Pending**, **Active**, or **Complete**.

**Tag** The tag value. If the message is being received with **MPI_ANY_TAG**, the display shows **ANY**.

**Type** The MPI data type (IBM MPI only). The MPI data type associated with the operation; for example, **MPI_INT()**.

**User Buffer, System Buffer, or Buffer**
The address of the buffer. Dive into this field to view a data window displaying the buffer contents.

## MPI Process Diving

To display more detail, you can dive into certain fields in the Message State Window. When you dive into a process field, TotalView does one of the following:

- Raises its Process Window if it exists.
- Sets the focus to an existing Process Window on the requested process.
- If a Process Window does not exist, creates a new one for the process.

If there is no relevant Process Window and you want TotalView to create a new Process Window instead of refocusing an existing Process Window, hold down the Shift key with the dive button.

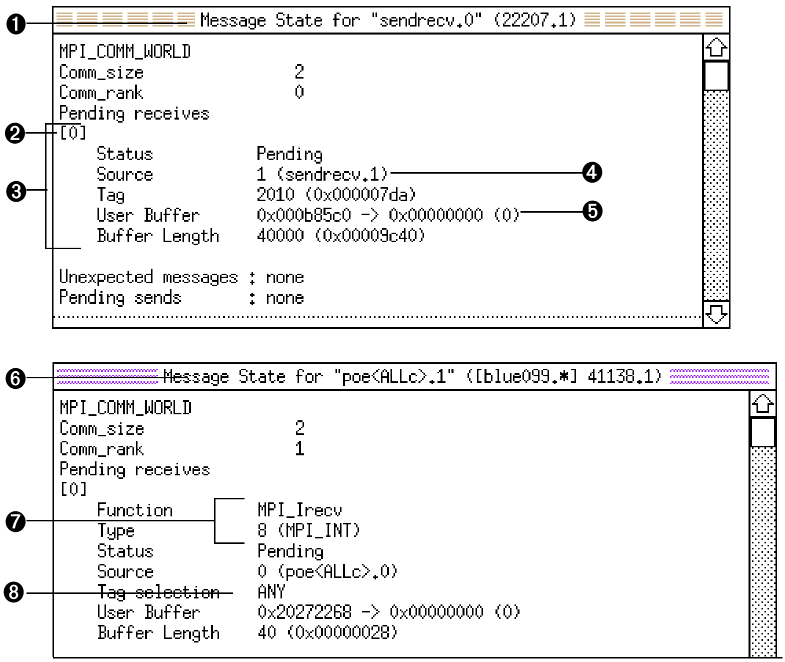
## MPI Buffer Diving

When you dive into the buffer fields, Totalview opens a data window. It also guesses what the correct format for the data should be based on the buffer's length and the data's alignment. If TotalView guesses incorrectly, you can edit the type field in the data window.

**NOTE** TotalView does not set the buffer type using the MPI data type.

## Pending Receive Operations

TotalView displays each pending receive operation in the **Pending receives** list. The following figure shows examples of MPICH and IBM MPI pending receive operations.




❶ MPICH
❷ Operation index
❸ One receive operation
❹ Dive to view process
❺ Dive to view data
❻ IBM MPI
❼ Additional fields
❽ Tag selection of ANY

FIGURE 37: **Message State Pending Receive Operation**

**NOTE** TotalView displays all of the receive operations that are maintained by the IBM MPI library. You should set the environment variable MP_EUIDEVELOP to the value DEBUG if you want blocking operations to be visible; otherwise, only non-blocking operations are maintained. For more details on the MP_EUIDEVELOP environment variable, consult the IBM *Parallel Environment Operations and Use* manual.

## Unexpected Messages

The Unexpected messages portion of the **Message State** Window shows information for messages that the MPI library has retrieved and enqueued, but which are not yet matched with a receive operation. Figure 38 shows an example of MPICH unexpected messages.

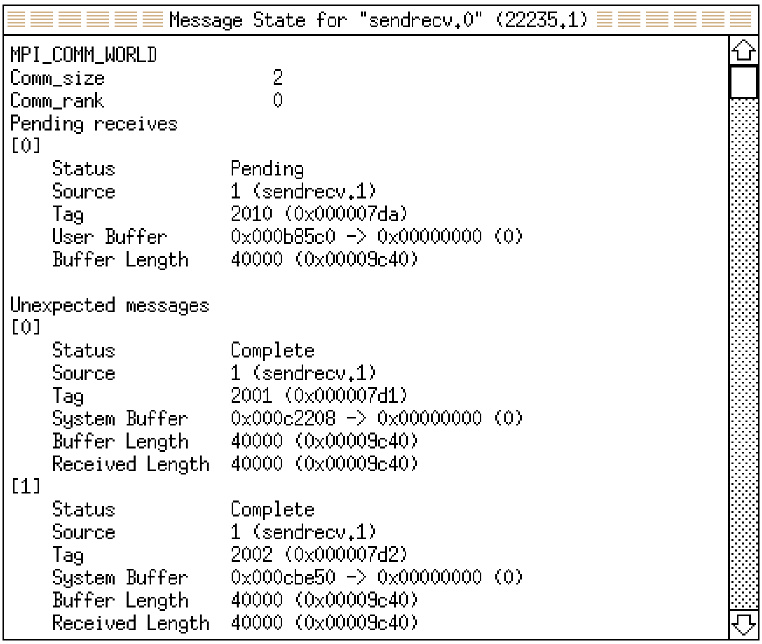
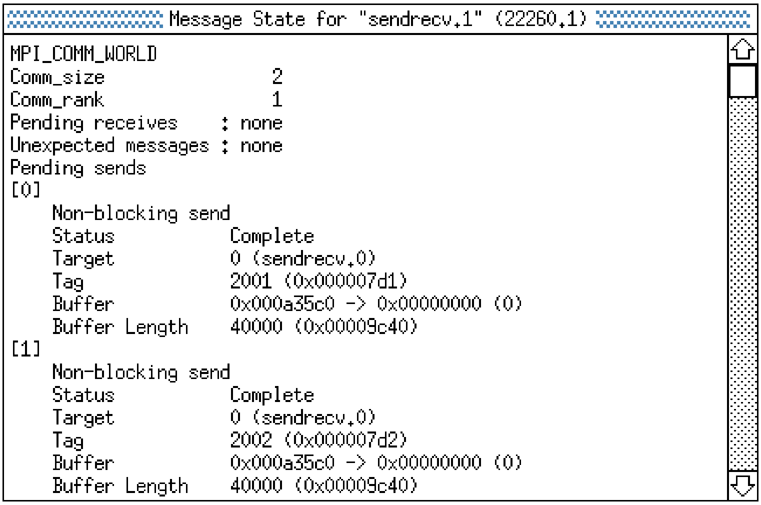



FIGURE 38: **Message State Unexpected Messages**

Some MPI libraries such as MPICH 1.1.1 only retrieve an already received message as a side effect of calls to functions such as **MPI_Recv()** or **MPI_Iprobe()**. (In other words, while some versions of MPI may know about the message, it may not as yet put it into a queue.) This means that the debugger can not list a message until after the destination process makes one of these kinds of calls.

## Pending Send Operations

TotalView displays each pending send operations in the **Pending sends** list. This is shown in Figure 39.



```
Message State for "sendrecv.1" (22260.1)
MPI_COMM_WORLD
Comm_size                 2
Comm_rank                 1
Pending receives    : none
Unexpected messages : none
Pending sends
[0]
    Non-blocking send
    Status           Complete
    Target           0 (sendrecv.0)
    Tag              2001 (0x000007d1)
    Buffer           0x000a35c0 -> 0x00000000 (0)
    Buffer Length    40000 (0x00009c40)
[1]
    Non-blocking send
    Status           Complete
    Target           0 (sendrecv.0)
    Tag              2002 (0x000007d2)
    Buffer           0x000a35c0 -> 0x00000000 (0)
    Buffer Length    40000 (0x00009c40)
```

FIGURE 39: **Message State Pending Send Operation**

MPICH does not normally keep information about pending send operations. However, when you configure MPICH, you can tell it to maintain a list of these operations. You will then need to start your program under TotalView's control and you use the **–ksq**, or **KeepSendQueue**, option to **mpirun**.)

Depending on the device for which MPICH was configured, blocking send operations may or may not be visible. However, if they are not displayed here, you can see that these operations are taking place because the call is in the stack backtrace.

If you attach to an MPI program that is not maintaining send queue information, TotalView displays the following message:

```
Pending sends : no information available
```

## MPI Debugging Troubleshooting

If you cannot successfully start TotalView on MPI programs, check the following:

- Can you successfully start MPICH programs without TotalView? The MPICH code contains some useful scripts to help you verify that you can start remote processes on all of the machines in your machines file. (See **tstmachines** in mpich/util.)
- Does the **tvdsvr** fail to start? **tvdsvr** must be on your PATH when you log in. Remember that **rsh** is being used to start the server, and it does not pass your current environment to the process you started remotely.
- You cannot get a message queue display if you get the following warning:

  ```
  The symbols and types in the MPICH library used by TotalView to
  extract the message queues are not as expected in the image
  <<your image name>>. This is probably an MPICH version or
  configuration problem.
  ```

  You need to check that you are using MPICH 1.1.0 or later and that you have configured it with the **–debug** option. (To verify this, look in the **config.status** file at the root of the MPICH directory tree).
- Make sure you have the correct MPI version and have applied the required patches. See the TOTALVIEW RELEASE NOTES for up-to-date information.
- Under some circumstances, MPICH kills TotalView with the **SIGINT** signal. You could see this behavior when restarting an MPICH job using the debugger's **Delete Program** command in the Process Window. If TotalView exits and is terminated abnormally with a **Killed** message, try setting the TotalView **–ignore_control_c** command line option. For example:

  ```
  setenv TOTALVIEW "totalview –ignore_control_c"
  mpirun –tv /users/smith/mympichexe
  ```

# Debugging OpenMP Applications

TotalView provides explicit support for many OpenMP C and Fortran compilers. The compilers and architectures that we support are listed in the TOTALVIEW RELEASE NOTES and our web site.

Here are some of the features that TotalView supports:

- Source level debugging of the original OpenMP code.
- The ability to plant breakpoints throughout the OpenMP code, including lines that are executed in parallel.
- Visibility of OpenMP worker threads.

- Access to SHARED and PRIVATE variables in OpenMP PARALLEL code.
- A stack back link token in worker threads' stacks so that you can find their master stack.
- Access to OMP THREADPRIVATE data in code compiled by the IBM and Compaq compilers.

The example code used in this section is included in the TotalView distribution in the file named **examples/omp_simple_f**.

**NOTE** On the SGI IRIX platform, you must use the MIPSpro 7.3 compiler or later to debug OpenMP.

## Debugging an OpenMP Program

Debugging an OpenMP code is very similar to debugging a multithreaded code, only differing in that the OpenMP compiler makes the following special code transformations:

- The most visible transformation is *outlining*. The compiler pulls the body of a PARALLEL region out of the original routine and places it into an *outlined routine*. In some cases, the compiler will generate multiple outlined routines from a single PARALLEL region. This allows multiple threads to execute the PARALLEL region.

  The outlined routine's name is based on the original routine's name.
- The compiler inserts calls to the OpenMP runtime library.
- The compiler splits variables between the original routine and the outlined routine. Normally, shared variables are maintained in the master thread's original routine, and private variables are maintained in the outlined routine.
- The master thread creates threads to share the work load. As the master thread begins to execute a parallel region in the OpenMP code, it creates the worker threads, dispatches them to the outlined routine, and then calls the outlined routine itself.

TotalView makes these transformations visible in the debugging session. Here are some things you should know:

- The compiler will generate multiple outlined routines from a single parallel region. This means that a single line of source code can generate multiple blocks of machine code inside different functions.

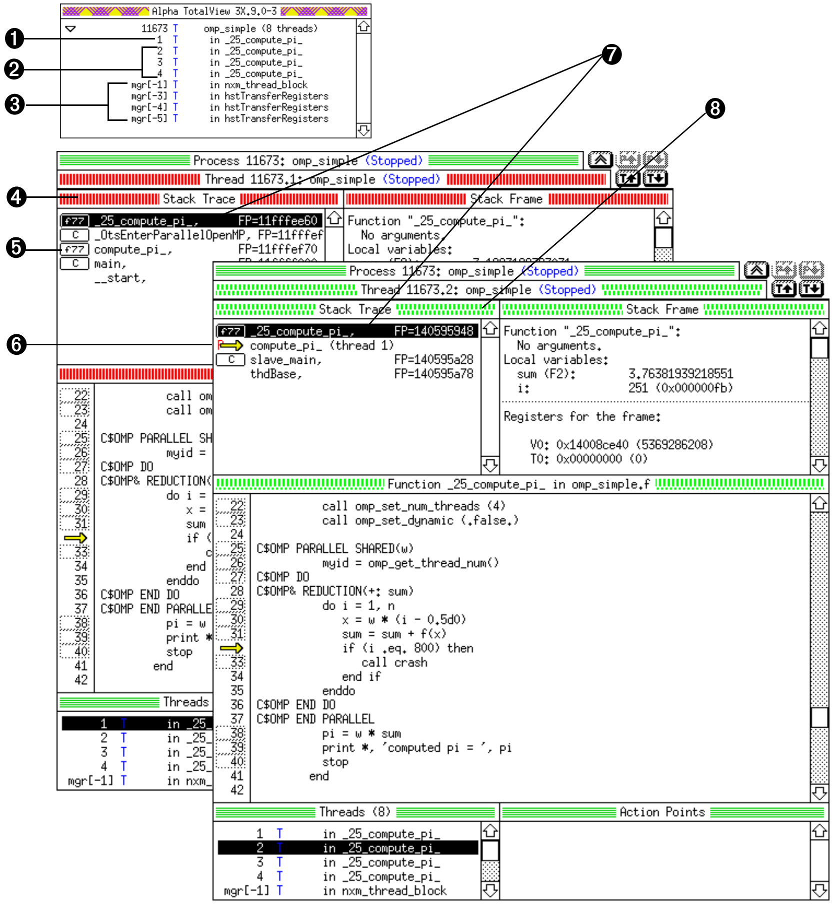
If you set a breakpoint on a source line that results in multiple outlined routines, TotalView asks you to differentiate the function name using the ambiguous source line selection dialog box. In most cases, you will select the **All** button to operate on all instances of the outlined functions.

- You cannot single step into or out of a parallel region. Instead, set a breakpoint inside the parallel region and allow the process to run to it. Once inside a parallel region, you can single step within it.
- OpenMP programs are multithreaded programs, so the rules for debugging multithreaded programs apply.

Figure 40 shows a sample OpenMP debugging session.

Notice the following:

- On Compaq Tru64 UNIX, the OpenMP threads are implemented by the compiler as **pthreads**, and on SGI IRIX as **sprocs**. TotalView shows the threads' logical and/or system thread ID, not the OpenMP thread number.
- The OpenMP master thread has logical thread ID number 1. The OpenMP worker threads have a logical thread ID number greater than 1.
- In Compaq Tru64 UNIX, the system manager threads have a negative thread ID; they do not take part in your OpenMP program, so do not touch them.
- SGI OpenMP uses the SIGTERM signal to terminate threads. Because TotalView stops a process when the process receives a SIGTERM, the OpenMP process will not be terminated. If you want the OpenMP process to terminate instead of stop, set the default action for the SIGTERM signal to *Resend*.
- When the OpenMP master thread is stopped in a PARALLEL DO outlined routine, the stack backtrace shows the following call sequence:
  - ➤ The outlined routine called from.
  - ➤ The OpenMP run time library called from.
  - ➤ The original routine (containing the parallel region).
- When the OpenMP worker threads are stopped in a PARALLEL DO outlined routine, the stack backtrace shows the following call sequence:
  - ➤ Outlined routine called from the special stack parent token line.
  - ➤ The OpenMP run time library called from.
- Select or dive on the stack parent token line to view the original routine's stack frame in the OpenMP master thread.




❶ OpenMP master thread
❷ OpenMP worker threads
❸ Manager threads (do not touch)
❹ Master thread Process Window
❺ "Original" routine name
❻ Stack parent token. Select or dive to view master
❼ "Outlined" routine name
❽ Worker thread process

FIGURE 40: **Sample OpenMP Debugging Session**

## OpenMP Private and Shared Variables

TotalView allows you to view both OpenMP private and shared variables.

OpenMP private variables are maintained in the outlined routine, and are stored by the compiler like local variables. See "*Displaying Local Variables and*

*Registers*" on page 143. However, OpenMP shared variables are maintained in the master thread's original routine stack frame.

TotalView allows you to display shared variables through a Process Window focused on the OpenMP master thread or through one of the OpenMP worker threads.

Here is how you display an OpenMP shared variable:

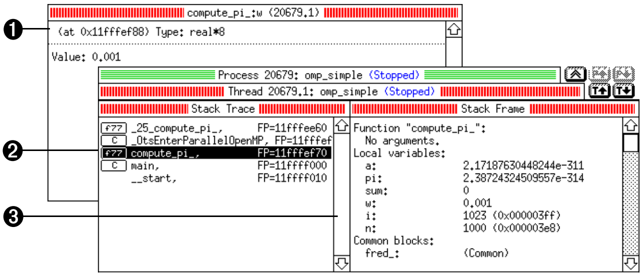
**1** Select the outlined routine in the Stack Trace Pane, or in the OpenMP master thread, select the original routine stack frame.

**2** Dive on the variable name, or display the **Function/File/Variable** menu and issue the **Variable...** command. When prompted, enter the variable name.

TotalView will open a Variable Window displaying the value of the OpenMP shared variable, as shown in Figure 41.

Shared variables are stored on the OpenMP master thread's stack. When displaying shared variables in OpenMP worker threads, TotalView uses the stack context of the OpenMP master thread to find the shared variable. TotalView uses the OpenMP master thread's context in the resulting Variable Window to display the shared variable.

You can also view OpenMP shared variables in the Stack Frame Pane by selecting the original routine stack frame in the OpenMP master thread, or by selecting the stack parent token line in the Stack Trace Pane of OpenMP worker threads, as shown in Figure 41.




❶ OpenMP shared variables have master thread's context
❷ Original routine's stack frame selected
❸ Stack Frame Pane includes shared variables

FIGURE 41: **OpenMP Shared Variable**

## OpenMP THREADPRIVATE Common Blocks

The Compaq Tru64 UNIX OpenMP and SGI IRIX compilers implement OpenMP THREADPRIVATE common blocks using the thread local storage system facility. This facility stores a variable declared in OpenMP THREADPRIVATE common blocks at different memory locations for each thread in an OpenMP process, which allows the variable to have different values in each thread.

To view a variable in an OpenMP THREADPRIVATE common block, or the OpenMP THREADPRIVATE common block itself, do the following:

1 In the Thread List Pane of the Process Window, select the thread containing the private copy of the variable or common block you would like to view.

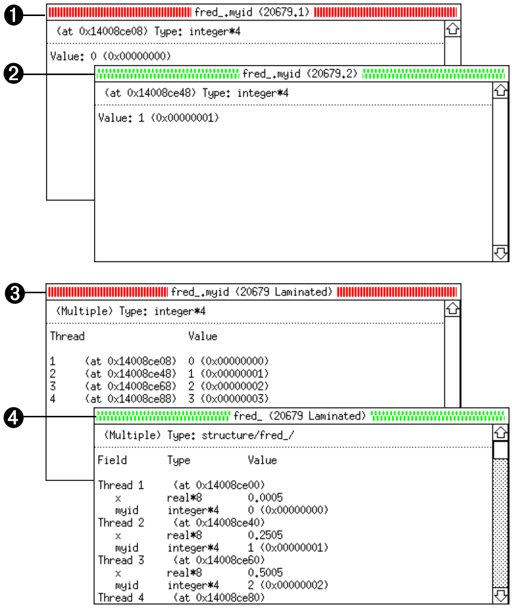
2 In the Stack Trace Pane of the Process Window, select the stack frame that will allow you to access OpenMP THREADPRIVATE common block variable. You can select either the outlined routine or the original routine for an OpenMP master thread. You must, however, select the outlined routine for an OpenMP worker thread.

3 From the Process Window, dive on the variable name or common block name. Or, display the **Function/File/Variable** menu and issue the **Variable...** command. When prompted, enter the name of the variable or common block. You may need to append an underscore (_) after the common block name. See "*Displaying Variable Windows*" on page 143 for more information on how to display variables.

   TotalView opens a Variable Window displaying the value of the variable or common block for the selected thread.

4 To view OpenMP THREADPRIVATE common blocks or variables across all threads, you can use the **Toggle Thread Laminated Display** command in the Variable Window. See "*Displaying a Variable in All Processes or Threads*" on page 180.

Figure 42 shows Variable Windows displaying OpenMP THREADPRIVATE variables and common blocks. Because the Variable Window has the same thread context as the Process Window from which it was created, the title bar patterns for the same thread match. In the laminated views, the values of the variable or common block across all threads are displayed.




❶ Variable from thread 1
❷ Variable from thread 2
❸ Thread laminated view of a single variable
❹ Thread laminated view of the entire common block

FIGURE 42: **OpenMP THREADPRIVATE Common Block Variables**

## OpenMP Stack Parent Token Line

TotalView inserts a special stack parent token line in the Stack Trace Pane of OpenMP worker threads when they are stopped in an outlined routine.

When you select or dive on the stack parent token line, the Process Window switches to the OpenMP master thread, allowing you to see the stack context of the OpenMP worker thread's routine. This context includes the OpenMP shared variables. (See Figure 43.)

You can select or dive on the OpenMP stack parent token line indicated by the PC arrow.

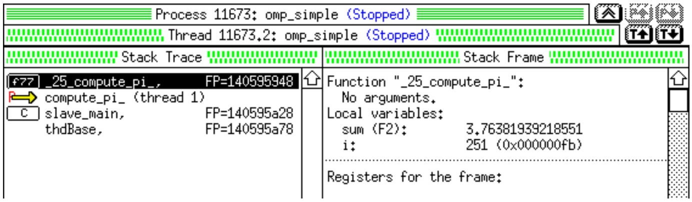



FIGURE 43: **OpenMP Stack Parent Token Line**

# Debugging PVM and DPVM Applications

You can debug applications that use the Parallel Virtual Machine (PVM) library or the Compaq Tru64 UNIX Parallel Virtual Machine (DPVM) library with Totalview on some platforms. TotalView supports ORNL PVM 3.3.4 or later on the Compaq Alpha, Hewlett-Packard, Sun 5, RS/6000, and SGI IRIX platforms and DPVM 1.4 or later on the Compaq Alpha platform.

**NOTE** See the TotalView Release Notes for the most up-to-date information regarding your PVM or DPVM software.

When you debug a PVM or DPVM application, TotalView becomes a PVM tasker, which establishes a debugging context for the duration of your session. You can run:

- One TotalView PVM or DPVM debugging session for a user and for an architecture; that is, different users cannot interfere with each other on the same machine or same machine architecture.

  One user can start TotalView to debug the same PVM or DPVM application on different machine architectures. However, a single user cannot have multiple instances of TotalView debugging the same PVM or DPVM session on a single machine architecture.

  For example, suppose you start a PVM session on Sun 5 and Compaq Alpha machines. You must start two TotalView sessions: one on the Sun 5 machine to debug the Sun 5 portion of the PVM session, and one on the Compaq Alpha machine to debug the Compaq Alpha portion of the PVM session. These two TotalView sessions are separate and do not interfere with one another.

- Similarly, in one TotalView session, you can run either a PVM application or a DPVM application, but not both. However, if you run TotalView on a Compaq Alpha, you can have two TotalView sessions, one debugging PVM and one debugging DPVM.

## Setting Up ORNL PVM Debugging

To enable PVM, create a symbolic link from the PVM **bin** directory (which is **$HOME/pvm3/bin/$PVM_ARCH/tvdsvr**) to the TotalView Debugger Server (**tvdsvr**).With this link in place, TotalView can use the **pvm_spawn()** call to spawn the **tvdsvr** tasks.

For example, if **tvdsvr** is installed in the **/opt/totalview/bin** directory, enter the following command:

```
ln -s /opt/totalview/bin/tvdsvr \
                    $HOME/pvm3/bin/$PVM_ARCH/tvdsvr
```

If the symbolic link does not exist, TotalView cannot spawn the debugger server and displays the following error:

```
Error spawning TotalView Debugger Server: No such file
```

## Starting an ORNL PVM Session

Start the ORNL PVM daemon process before you start TotalView. See the ORNL PVM documentation for information about the PVM daemon process and console program.

1. Use the **pvm** command to start a PVM console session, which will start the PVM daemon. If PVM is not running when you start TotalView (with PVM support enabled), TotalView exits with the following message:

   Fatal error: Error enrolling as PVM task: *pvm error*
2. If your application uses groups, start the **pvmgs** process before starting TotalView. PVM groups are unrelated to TotalView process groups. For information about TotalView process groups, refer to "*Examining Process Groups*" on page 124.
3. Enable PVM support in TotalView using an X resource; see "TOTALVIEW*PVMDEBUGGING" on page 288. You need to restart TotalView after setting this new resource. For more information, refer to "*X Resources*" on page 275.

As an alternative, you can use command line options to the **totalview** command. For example:

**–pvm** *which enables* PVM *support*
**–no_pvm** *which disables* PVM *support*

The command-line options override the X resource. For more information on the **totalview** command, refer to "*TotalView Command Syntax*" on page 299.

4 Set the TotalView directory search path to include the PVM directories. This directory list must include those needed to find both executable and source files. The directories you use will vary, but should always contain the current directory and your home directory.

You can set the directory search path using an X resource or the **Set Search Directory** command. Refer to "TOTALVIEW*SEARCHPATH" on page 289 and "*Setting Search Paths*" on page 44 for more information.

For example, to debug the PVM examples, you can specify the following list of directories in your search path:

```
$HOME
$PVM_ROOT/xep
$PVM_ROOT/xep/$PVM_ARCH
$PVM_ROOT/src
$PVM_ROOT/src/$PVM_ARCH
$PVM_ROOT/bin/$PVM_ARCH
$PVM_ROOT/examples
$PVM_ROOT/examples/$PVM_ARCH
$PVM_ROOT/gexamples
$PVM_ROOT/gexamples/$PVM_ARCH
```

5 Verify that the action taken by TotalView for the SIGTERM signal is appropriate. (You can examine the current action using the **Set Signal Handling Mode** command. Refer to "*Handling Signals*" on page 41 for more information.)

PVM uses the SIGTERM signal to terminate processes. Because TotalView stops a process when the process receives a SIGTERM, the OpenMP process is not terminated. If you want the PVM process to terminate, set the action for the SIGTERM signal to **Resend**.

Continue with "PVM/DPVM *Automatic Process Acquisition*" on page 100.

## Starting a DPVM Session

DPVM requires no additional user configuration. However, you must start the DPVM daemon before you start TotalView. See the DPVM documentation for information about the DPVM daemon and console program.

**1** Use the **dpvm** command to start a DPVM console session; starting the session also starts the DPVM daemon. If DPVM is not running when you start TotalView (with DPVM support enabled), TotalView exits with the following message:

```
Fatal error: Error enrolling as DPVM task: dpvm error
```

**2** You can enable Enable DPVM support in two ways. The first uses an X resource; see "TOTALVIEW*DPVMDEBUGGING" on page 281. You'll need to restart TotalView after setting (or resetting) an X resource.

As an alternative, you can use command line options to the **totalview** command. For example:

**–dpvm** *which enables* DPVM *support*.
**–no_dpvm** *which disables* DPVM *support*

The command-line options override the X resource. For more information on the **totalview** command, refer to "*TotalView Command Syntax*" on page 299.

**3** Verify that the default action taken by TotalView for the SIGTERM signal is appropriate. You can examine the default actions with the **Set Signal Handling Mode** command in TotalView. Refer to "*Handling Signals*" on page 41 for more information.

DPVM uses the SIGTERM signal to terminate processes. Because the debugger stops a process when the process receives a SIGTERM, the OpenMP process is not terminated. If you want the DPVM process to terminate, set the action for the SIGTERM signal to **Resend**.

If you enable PVM support using X resources, and you wish to use DPVM, you must use both

**–no_pvm** and **–dpvm** command line options when you start TotalView. Similarly, when enabling DPVM support with X resources, use the **–no_dpvm** and **–pvm** command line options to debug PVM.

**NOTE** Do not use X resources to start both PVM and DPVM.

## PVM/DPVM Automatic Process Acquisition

This section describes how TotalView automatically acquires PVM and DPVM processes in a PVM or DPVM debugging session. Specifically TotalView uses the PVM tasker feature to intercept **pvm_spawn()** calls.

When you start TotalView as part of a PVM or DPVM debugging session, it takes the following actions:

- TotalView checks to make sure there are no other PVM or DPVM taskers running. If TotalView finds a tasker on any host that it is debugging, it displays the following message and then exits:

  `Fatal error: A PVM tasker is already running on host 'host'`
- TotalView finds all the hosts in the PVM or DPVM configuration. Using the **pvm_spawn()** call, TotalView starts a TotalView Debugger Server (**tvdsvr**) on each remote host that has the same architecture type as the host on which TotalView is running. It tells you it has started a debugger server by printing:

  `Spawning TotalView Debugger Server onto PVM host 'host'`

If you add a host with a compatible machine architecture to your PVM or DPVM debugging session after you start TotalView, TotalView automatically starts a debugger server on that host.

After all debugger servers are running, TotalView will intercept every PVM or DPVM task created using the **pvm_spawn()** call on hosts that are part of the debugging session. If a PVM or DPVM task is created on a host with a different machine architecture, TotalView ignores that task.

When TotalView receives a PVM or DPVM tasker event, it takes the following actions:

1. TotalView reads the symbol table of the spawned executable.
2. If a saved breakpoints file for the executable exists and you have automatic loading of breakpoints enabled, TotalView loads breakpoints for the process.
3. TotalView asks if you want to stop the process before it enters the **main()** routine.

   If you answer **Yes**, TotalView stops the process before it enters **main()** (that is before it executes any user code). This allows you to set breakpoints in the spawned process before any user code executes. On most machines, TotalView stops a process in the **start()** routine of the **crt0.o** module if it is statically linked. If the process is dynamically linked, the debugger stops it just after it finishes running the dynamic linker. Because the Process Window displays assembler instructions, you will need to use the **Function or File** command to display the source code for the **main()** routine. For more information on this command, refer to "*Finding the Source Code for Functions*" on page 115.

## Attaching to PVM/DPVM Tasks

You can attach to a PVM or DPVM task if the task meets the following criteria:

- The machine architecture on which the task is running is the same as the machine architecture on which TotalView is running.
- The task must be created. (This is indicated when flag 4 is set in the PVM Tasks and Configuration Window.)
- The task must not be a PVM tasker. If flag 400 is clear in the PVM Tasks and Configuration Window, the process is a tasker.
- The executable name must be known. If the executable name is listed as a dash (–), TotalView cannot determine the name of the executable. (This can occur if a task was not created using the **pvm_spawn()** call.)

To attach to a PVM or DPVM task, complete the following steps:

**1** Issue the **Show All PVM Tasks** command from the TotalView Root Window.

The **PVM Tasks And Configuration** Window is displayed, as shown in Figure 44. This window displays current information about PVM tasks and hosts—TotalView automatically updates this information as it receives events from PVM.

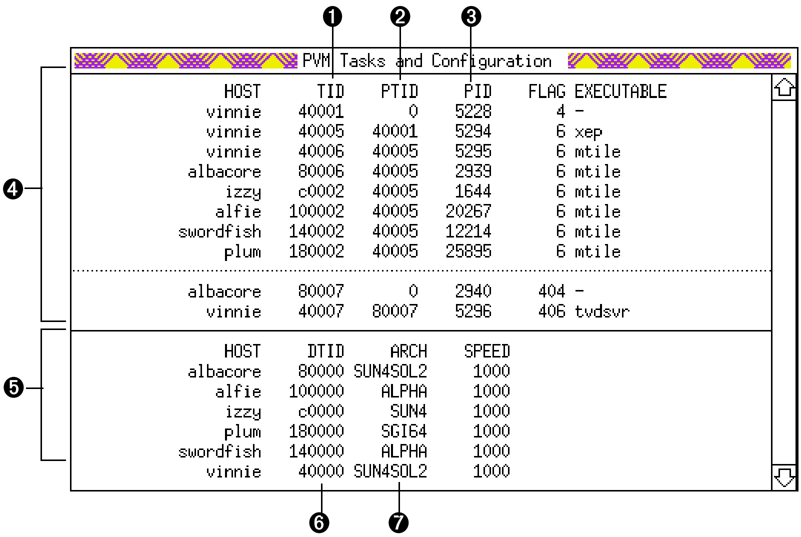
Since PVM does not always generate an event that allows TotalView to update this window, you should use the **Update PVM Task List** command to update it when you need current information.

For example, you can attach to the tasks named **xep** and **mtile** in the following figure because flag 4 is set. In contrast, you cannot attach to the **tvdsvr** and **–** executables because flag 400 is set.

**2** Dive on a task entry that meets the criteria for attaching to tasks. TotalView attaches to the task.

**3** If the task to which you attached has related tasks that can be debugged, TotalView asks if you want to attach to these related tasks. If you answer **Yes**, TotalView attaches to them. If you answer **No**, it only attaches to the task you dove on.

After attaching to a task, TotalView looks for attached tasks that are related to the this task; if there are related tasks, TotalView places them in the same program group. If TotalView is already attached to a task you dove on, it simply opens and raises the Process Window for the task. (See Figure 44.)




❶ Task ID (TID)
❷ Parent TID
❸ UNIX Process ID (PID)
❹ Tasks
❺ Hosts
❻ Daemon TID
❼ Machine Architecture

FIGURE 44: **PVM Tasks and Configuration Window**

**Reserved Message Tags:** TotalView uses PVM message tags in the range 0xDEB0 through 0xDEBF to communicate with PVM daemons and the TotalView Debugger Server. Avoid sending messages that use these reserved tags.

**Debugging Dynamic Libraries:** If the machines in your PVM debugging session are running different versions of the same operating system, the dynamic libraries can vary from machine to machine. If this is the case, you may see strange stack backtrace results when your program is executing inside a dynamic library. To eliminate this problem, make sure all of the hosts in your PVM configuration are running the same version of the operating system and have the same dynamic libraries installed. As an alternative, you can statically link your programs.

**Cleanup of Processes:** The **pvmgs** process registers its task ID in the PVM database. If the **pvmgs** process is terminated, the **pvm_joingroup()** routine hangs because PVM does not clean up the database. If this happens, you must terminate and then restart the PVM daemon.

TotalView attempts to clean up the TotalView Debugger Server daemons (**tvdsvr**), which also act as taskers. If some of these processes do not terminate, you must manually terminate them.

# Shared Memory Code

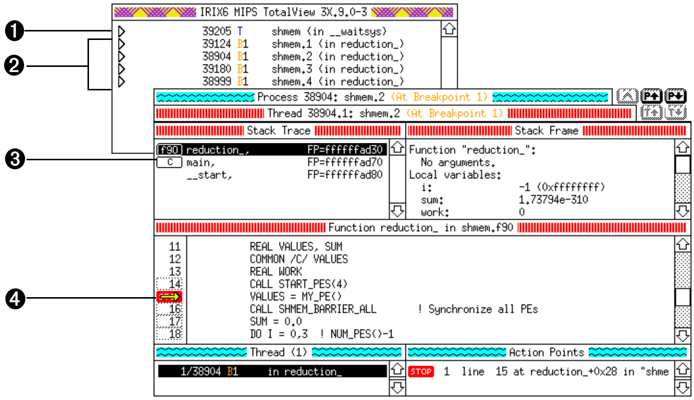
TotalView supports the SGI IRIX logically shared, distributed memory access (SHMEM) library.

To debug a SHMEM program, follow these steps:

1. Link it with the **dbfork** library. See "*Linking with the dbfork Library*" on page 324.
2. Start TotalView on your program. See Chapter 3, "*Setting Up a Debugging Session*" on page 29.
3. Set at least one breakpoint after the call to the **start_pes()** SHMEM routine. (This is illustrated in the following figure.)

The call to **start_pes()** creates new worker processes that return from the **start_pes()** call and execute the remainder of your program. The original process never returns from **start_pes()**, but instead stays in that routine, waiting for the worker processes it created to terminate.

**NOTE** You cannot single-step over the call to start_pes().




❶ SHMEM starter process
❷ SHMEM worker processes
❸ Select a worker process in the Root Window
❹ Set breakpoint *after* the call to start_pes()

FIGURE 45: **SHMEM Sample Session**

# Debugging Portland Group, Inc. HPF Applications

TotalView allows the source level debugging of High Performance Fortran (HPF) code compiled with the Portland Group Inc. HPF (PGI HPF) compiler.

**NOTE** Debugging PGI HPF programs requires a separate TotalView license.

TotalView supports the following platforms:

- IBM RS/6000 and SP AIX 4.x
- SGI MIPS IRIX 6.x, for programs compiled with –64 only
- Sun Sparc SunOS 5 (Solaris 2.x)

See the TOTALVIEW RELEASE NOTES for supported PGI HPF runtime configurations.

In addition to normal TotalView features, the TotalView PGI HPF support allows:

- Source level display of HPF code.
- Source level breakpoints in HPF code.

- You can update replicated scalar variables in all processes by updating the value in any process. If the values were not all the same at the start, TotalView gives you a warning, and you have to explicitly agree to the update before it will take place.
- Display of distributed arrays, with optional display of the owning processor.
- Display of the distribution of distributed arrays, for instance, onto which node a particular element of a distributed array has been mapped.
- Visualization of distributed arrays.
- Automatic update of all copies of replicated scalar variables.

However, there are still a number of limitations:

- Display of user defined data types is not yet supported.
- Evaluation points and expressions are executed locally and cannot reference distributed arrays. However, you can use the **$visualize** intrinsic.

If you use the **$visualize EVAL** intrinsic, remember that **EVAL** code is executed by every process. Therefore, you probably want to make this an non-shared action point.

- You can export the distribution of an array to the visualizer to display it graphically.
- You see the HPF source and variables.
- You can set breakpoints in the HPF source code.

In the address display for data windows showing HPF variables, there is an additional field which tells you whether the variable is distributed [**Dist**] or replicated [**Repl**]. If you update a replicated variable, it is updated in all processes. A distributed variable is only updated in its home process.

You cannot edit the address of a distributed array. If you edit the address of a replicated scalar, it will be marked as distributed, since it no longer makes sense to update all of the processes, as you do not know what is at that address in the other processes.

When you display an HPF distributed array, TotalView can also display the logical processor on which each element resides. The display of this additional information can be changed for a single data window using the **Toggle Node Display** option in the menu of the data window. You can set the default for a whole TotalView session by using the command line options **–hpf_node** or **–no_hpf_node**; you could also use the X resource "TOTALVIEW*HPFNODE" ON PAGE 283. No matter which way you set the default, you can always toggle the behavior in each window.

By default, this display is disabled. If it is enabled, a distributed array will look like Figure 46. Otherwise, the Node column is not displayed and a distributed array display looks the same as that of a normal array.

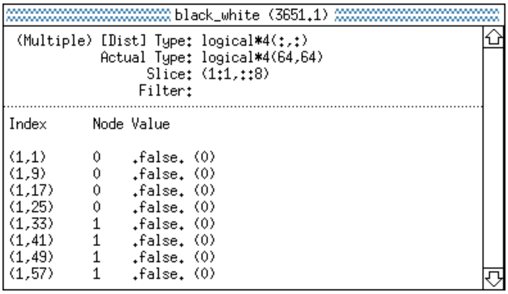



FIGURE 46: **Block Distributed Array on Three Processes**

To see the distribution of an array or a section of an array, use the **Visualize Distribution** command from the data window menu. This command exports the HPF processor number on which each selected element of the array resides to the visualizer. This command differs from the **Visualize** command in that it exports the values of the array elements, not the ownership information.

This capability is not available with the **$visualize** command since distributions are normally static.

## Starting TotalView with HPF

Beginning with PGI HPF release 2.4, TotalView can track a process started by **rpm** or **smp**, the default PGI HPF runtime libraries. If you still want to use MPI, then you need to ensure that the MPI implementation is supported by PGI HPF and TotalView. See "*Debugging MPI Applications*" on page 69.

On IBM SP, or clusters of RS/6000 machines running IBM's Parallel Environment, you can use any runtime library that is started using the **poe** command.

On SGI IRIX, TotalView supports 64-bit PGI HPF programs only. You must compile your PGI HPF program with the **–64** compiler option.

## Dynamically Loaded Library

To debug PGI HPF code, TotalView needs to be able to dynamically load the file **libtvhpf.so**, which is distributed as part of the PGI HPF product.

TotalView searches for this file in the following order:

1 TotalView attempts to dynamically load the unadorned file name **libtvhpf.so**. This will succeed if **libtvhpf.so** is in one of the directories on your dynamic library path environment variable (on Sun Sparc SunOS5, IBM AIX, and SGI IRIX, this is LD_LIBRARY_PATH if the variable LD_LIBRARYN32_PATH is not set). On SGI IRIX, **libtvhpf.so** is in one of the directories on your **–n32** dynamic loader path (LD_LIBRARYN32_PATH).

2 If step 1 fails, TotalView uses the **PGI** environment variable to find the Portland Group installation. If the **PGI** environment variable is not set, TotalView looks for the default installation directory (**/usr/pgi**).

3 Depending on your architecture, TotalView then searches the directories in the order shown in the following table.

TABLE 9: **PGI HPF Dynamic Library Search Order**

| System | Search Path |
|---|---|
| IBM RS/6000 and SP AIX 4.x | $PGI/sp2/lib |
| | $PGI/rs6000/lib |
| Sun Sparc SunOS 5 (Solaris 2.x) | $PGI/solaris/lib |
| SGI MIPS IRIX 6.x | $PGI/sgi/lib–n32 |
| | $PGI/sgi/lib–64 |
| | $PGI/origin/lib/mips4 |

If TotalView still cannot locate a copy of **libtvhpf.so** and, if the TotalView verbosity level is not **silent**, an error message is displayed telling you that the library could not be found, **HPF** debugging is disabled. TotalView will then start debugging the generated Fortran code.

If TotalView cannot find your copy of **libtvhpf.so**, you should either move it to one of the places that will be searched by default, or add its directory to your LD_LIBRARY_PATH.

## Setting Up PGI HPF Compiler Defaults

**NOTE** With PGI HPF version 2.4 and later, there is no need to use an MPICH based runtime, and you can ignore this section.

Set up the HPF compiler with the defaults set for using MPICH, TotalView, the IBM parallel environment, and FORTRAN 77, as in the following sections.

If you have PGI HPF release 2.4, the **rc** files should already have been set up correctly, but they will use the default runtime, which is not MPI. If you want to use an MPI runtime, you should consult the PGI HPF manuals.

## Setting Up MPICH

You should follow the instructions in the PGI HPF manual and MPICH manual to ensure that you can build an HPF program and run it using MPICH. One way to do this is to create your own **.pghpfrc** file and add lines similar to the following:

```
#   Set up to use my MPI with PGI HPF.
#   Change the path to libmpi.a as appropriate
#
INCLUDE  $DRIVER/.pghpfrc
set HPF_MPI=/where_your_mpi_lives/libmpi.a
set HPF_COMM_LIBS= \
            "–lpghpf_mpi$P  $HPF_MPI  $HPF_SOCKET"
```

Because these lines tell **pghpf** to use the MPI communications library, you do not need to name them on the command line at compilation time.

## Setting TotalView Defaults for HPF

To debug HPF code, you will normally set the breakpoint and barrier breakpoint behavior so that TotalView does not stop other processes when the breakpoint is hit. For more information, refer to "*Parallel Debugging Tips*" on page 110.

Other HPF resources are "TOTALVIEW*HPF" on page 283 and "TOTALVIEW*HPFNODE" ON PAGE 283.

## Compiling HPF for Debugging

To compile your HPF PROGRAM so it can be used with TotalView, you should use the **–g** and **–Mtotalview** options to **pghpf** when both compiling and linking. (The **–Mtv** option is the same as the **–Mtotalview** option.)

The **–g** option can produce confusing results when used by itself. For example, while you may see the HPF source code, none of the HPF debugging features will work. If TotalView flags your HPF code in the stack backtrace as being **f77**, the program was probably not compiled with the **–Mtv** option.

If you want to debug the Fortran code generated by HPF, you must also use the **–Mkeepftn** option. Otherwise, the compiler deletes these intermediate Fortran files after it compiles the source code.

You can debug at the generated Fortran level by starting TotalView with the **–no_hpf** option or setting the X resource **totalview*hpf** to **false**. TotalView will then ignore the **.stb** and **.stx** files and show you the generated F77.

There is no need to relink the HPF program to debug at the generated Fortran level.

## Starting HPF Programs

The way in which TotalView starts an HPF parallel program depends on the machine on which the code is running and the run time library linked into the HPF code.

### PGI HPF smp and rpm libraries

Using TotalView to start a program linked with the **smp** and **rpm** libraries is similar to the way in which you would normally start the program. For example, suppose you would start the program as follows:

```
my_program –bah –pghpf –np 6
```

Here is the command you would use to debug it using TotalView:

```
totalview my_program –a –bah –pghpf –np 6
```

### Starting HPF Programs with MPICH

In a workstation cluster environment using MPICH, debug your HPF application with TotalView by adding the **–tv** option to the **mpirun** command. For example, assume that you would begin executing your code with the following command:

```
mpirun –np 4 my_program
```

Using **mpirun**, you would invoke TotalView as follows:

```
mpirun –tv –np 4 my_program
```

### Workstation Clusters Using MPICH

Debugging workstation clusters uses the same mechanism as debugging an MPICH program since a compiled HPF program is an MPICH program. For more information, refer to "*Debugging MPI Applications*" on page 69.

### IBM Parallel Environment

In the IBM parallel environment on an IBM SP or cluster of RS/6000 machines, parallel programs are started with the **poe** command. To debug parallel codes, you invoke TotalView on the **poe** command, for instance:

```
totalview poe –a hpf_test –procs 6
```

For more information, refer to "*Starting TotalView on a PE Job*" on page 77.

# Parallel Debugging Tips

When you are debugging your parallel programs, the following points are important to remember.

## General Parallel Debugging Tips

Here are some tips that are useful for debugging most parallel programs:

- **Breakpoint behavior**

  When you are debugging message-passing and other multiprocess programs, it is usually easier to understand the program's behavior if you change the default stopping action of breakpoints and barrier breakpoints. By default, when one process in a multiprocess program hits a breakpoint, TotalView will stop all the other processes.

  To change the default stopping action of breakpoints and barrier breakpoints, you can set the X resources "TOTALVIEW*STOPALL" on page 292 and "TOTALVIEW*BARRIERSTOPALL" on page 278 to **false**.

  A second method is to specify the **–no_stop_all** TotalView command line options described on page 309 and **–no_barr_stop_all** described on page 301.

  These settings set breakpoint and barrier breakpoint behavior to allow other processes to continue to run when one process in a group hits the breakpoint.

  These options only affect the default behavior. As usual, you can choose a behavior for a breakpoint by setting the breakpoint properties in the action points dialog box. See "*Breakpoints for Multiple Processes*" on page 203.

- **Process synchronization**

  TotalView has two features that make it easier to get all of the processes in a multiprocess program synchronized and executing the line.

  Process barrier breakpoints and the process hold/release features work together to help you get control the execution of your processes. See "*Process Barrier Breakpoints*" on page 206.

  The **Run (to selection) Group** command is a special kind of stepping command. It allows you to run a group of processes to a selected source line or instruction. See "*Group-level Single Stepping*" on page 131.

- **Using group commands**

  Group commands are often more useful than process commands.

  It is often more useful to issue the **Go Group** command from the **Go/Halt/Step/Next/Hold** menu to restart the whole application, rather than use the **Go Process** command and then use the **Halt Group** command rather than the **Halt Process** command.

  The group-level single-stepping commands such as **Step Group** and **Next Group** allow you to single-step a group of processes in a parallel. See "*Group-level Single Stepping*" on page 131.

- **Process-level stepping**

  If you use a process-level single-stepping command in a multiprocess program, TotalView may appear to be hung (it continuously displays the watch cursor). If you single-step a process over a statement that cannot complete without allowing another process to run and that process is stopped, the stepping process appears to hang. In parallel programs, this can occur when you try to single-step a process over a communication operation that cannot complete without the participation of another process. When this happens, you can abort the single-step operation by pressing **Ctrl-C** in any TotalView window. As an alternative, consider using a group-level single-step command instead.

- **Determining which processes and threads are executing**

  The TotalView Root Window helps you determine where various processes and threads are executing. When you select a line of code in the Process Window, the Root Window is updated to give you visual feedback about which processes and threads are executing that line. See "*Displaying Thread and Process Locations*" on page 138.

- **Viewing variable values**

  You can view the value of a variable that is replicated across multiple processes or multiple threads in a single Variable Window. See "*Displaying a Variable in All Processes or Threads*" on page 180.

- **Restarting**

  You can restart a parallel program at any time during your debugging session. If your program runs too far, you can kill the program by displaying the **Arguments/Create/Signal** menu in the Process Window and selecting the **Delete Program** command. This command kills the master process and all the slave processes. Restarting the master process (for example, **mpirun** or **poe**) recreates all of the slave processes. Start-up is faster when you do this because TotalView does not need to reread the symbol tables or restart its server processes as they are already running.

## MPICH Debugging Tips

Here are some debugging tips that apply only to MPICH:

- **Passing options to mpirun**

  You can pass options to TotalView through the MPICH **mpirun** command.

  To pass options to TotalView when running **mpirun**, you can use the **TOTALVIEW** environment variable. For example, you can cause **mpirun** to invoke TotalView with the **–no_stop_all** option as in the following C-shell, example:

```
setenv TOTALVIEW "totalview –no_stop_all"
```

- **Using ch_p4**

  If you start remote processes with MPICH/**ch_p4**, you may need to change the way TotalView starts the servers.

  By default, TotalView uses **rsh** to start its remote server processes. This is the same behavior as **ch_p4**. If you configure MPICH/**ch_p4** to use a different start-up mechanism from another process, you will probably also need to change the way that TotalView starts the servers.

  For more information about **tvdsvr** and **rsh**, see "*Single Process Server Launch Options*" on page 55. For more information about **rsh**, see "*Single Process Server Launch Command*" on page 59.

## IBM PE Debugging Tips

Here are some debugging tips that apply only to IBM MPI (PE):

- **Avoid unwanted timeouts**

  You can cause undesired timeouts if you place breakpoints that stop other process too soon after calling **MPI_Init()** or **MPL_Init()**. If you create "stop all" breakpoints, it causes the first process to get to the breakpoint to stop all the other parallel processes that have not yet arrived at the breakpoint. This may cause a timeout.

  To turn the option off, click with the right mouse button on the **stop** symbol for the breakpoint. The breakpoint dialog box will come up, in which you should deselect the box labeled "Stop All Related Processes when Breakpoint Hit."

- **Control the poe process**

  Even though the **poe** process continues under TotalView control, you should not attempt to start, stop, or otherwise interact with **poe**. The parallel tasks require that **poe** continue to run for normal functioning. For this reason, if **poe** had been stopped, TotalView automatically continues it when you continue any of the parallel tasks.

- **Avoid slow processes due to node saturation**

  If you try to debug a Parallel Environment for AIX program in which more than three parallel tasks are run on a single node, the parallel tasks on each such node may run noticeably slower than they would run if you weren't debugging them.

This effect becomes more noticeable as the number of tasks increases, and, in some cases, the parallel tasks may make hardly any progress. This is because the Parallel Environment for AIX uses the SIGALRM signal to implement the communications operations, and the debugging interface in AIX requires that the debugger intercept all signals. As the number of parallel tasks on a node increases, the copy of TotalView or the TotalView Debugger Server running on that node becomes saturated, and cannot keep up with the SIGALRMs being sent, thus slowing down the tasks.

Chapter 6

# Debugging Programs

This chapter explains how to perform basic debugging tasks with TotalView. This chapter explains how you:

- Find code as you are debugging
- Display your code in source and assembler formats
- Return to the currently executing line in the stack frame
- Invoke your editor on source files you are debugging
- Interpret status and control registers
- Use commands for controlling processes and threads
- Control process groups in multiprocess programs
- Set action points
- Use single-step commands
- Set the program counter

## Finding the Source Code for Functions

You can search for the source code for any function in your program by selecting the **Function or File** command from the **Function/File/Variable** menu. Within the displayed dialog box, type the function name. (See Figure 47 on page 116.)

After TotalView finds the source code, it displays it in the Source Code Pane. If the function you selected was not compiled with source line information, TotalView displays disassembled machine code.

**NOTE** When you want to return to the previous contents of the Source Code Pane, use the undive icon located in the upper right corner of the source pane.

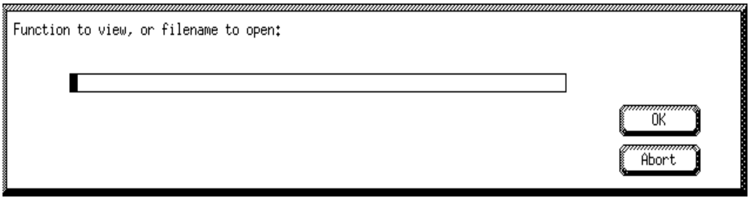



FIGURE 47: **Function Name Dialog Box**

You can use the **Edit Source Text** command (see "*Editing Source Text*" on page 121 for details) or an X Window System client such as **xmore**, **vi**, or **emacs** to display these files while debugging.

Another method of locating a function's source code is by diving into its from within the Source Code Pane.

## Resolving Ambiguous Names

Sometimes the function name you specify is ambiguous. For example, you may have specified the name of a static function and your program contains multiple static functions by that same name. Alternatively, you may have specified the name of a member function in a C++ program and there are multiple classes with member functions of that name. Or, you may have specified the name of an overloaded function or a template function. Figure 48 shows an example of the dialog that TotalView displays when it encounters an ambiguous function name.

To resolve the ambiguity, click one of the radio buttons or the text following it and then click **OK**. Alternately, you may type an unambiguous name in the **Function specification** field.

When you select a function name, its specification appears in the **Function specification** field. This allows you to enter a new function specification by editing an existing one. When there are many screens of function names in the dialog, this feature lets you specify a name without having to scroll to find it.

Because TotalView remembers the resolved specification, you do not need to select it again the next time you dive into the function.

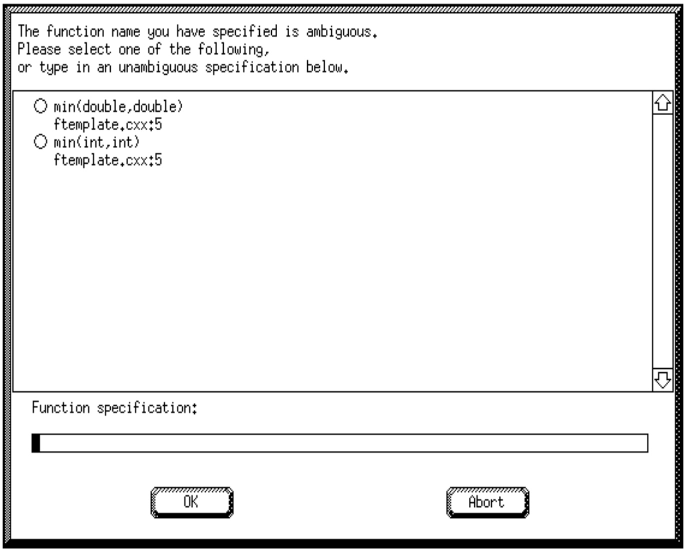



FIGURE 48: **Resolving Ambiguous Function Names Dialog Box**

TotalView may request that you set the context when you:

- Specify a function name with the **Function or File** command.
- Dive on a name in the Source Code Pane.
- Set a breakpoint at a line in the function.
- Select a function by clicking on its line in the Stack Trace Pane.

# Finding the Source Code for Files

You can display the source code for a given file in your program by choosing the **Function/File/Variable** menu and selecting the **Function or File** command. When prompted, enter the file name in the dialog box shown in Figure 47. You may enter the name of a header file if the header file contains source lines that produce executable code.

**NOTE** If TotalView determines that a file contains Fortran 90 code, functions, or subroutines defined earlier in the same source file may appear to be written in FORTRAN 77. This should not be a problem since these functions cannot be using Fortran 90 features.

# Examining Source and Assembler Code

You can display your program in several different ways. If you display assembler in the Source Code Pane, you can also display addresses in two different ways, as shown in Table 11.

TABLE 10: Ways to Display Source and Assembler Code

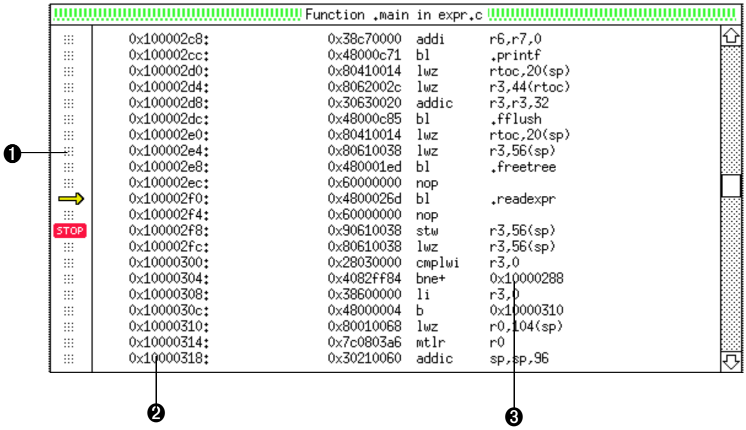
| To Display | Select This from the Display/Directory/ Edit menu ... |
|---|---|
| Source code (Default) | **Source Display Mode** |
| Assembler code | **Assembler Display Mode** |
| Source and assembler interleaved | **Interleave Display Mode**. Source statements are treated as comments. You can set breakpoints or evaluation points only at the machine level. Setting an action point at the first instruction after a source statement, however, is equivalent to setting a point at that source statement. |

You can tell TotalView to display assembler code using symbolic or absolute addresses, as described in the following table:

TABLE 11: Assembler Code Display Styles

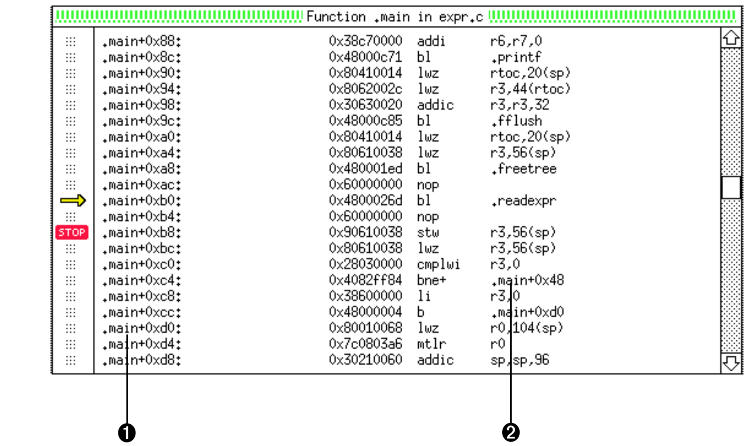
| To Display Address Using | Select This from the Display/Directory/ Edit menu ... |
|---|---|
| Absolute addresses for locations and references (Default) | **Display Assembler by Address** |
| Symbolic addresses (function names and offsets) for locations and references | **Display Assembler Symbolically** |

The following three figures illustrate the effect of displaying assembler code in different ways in the Source Code Pane. You can also display assembler instructions in a Variable Window. For more information, see "*Displaying Machine Instructions*" on page 147.



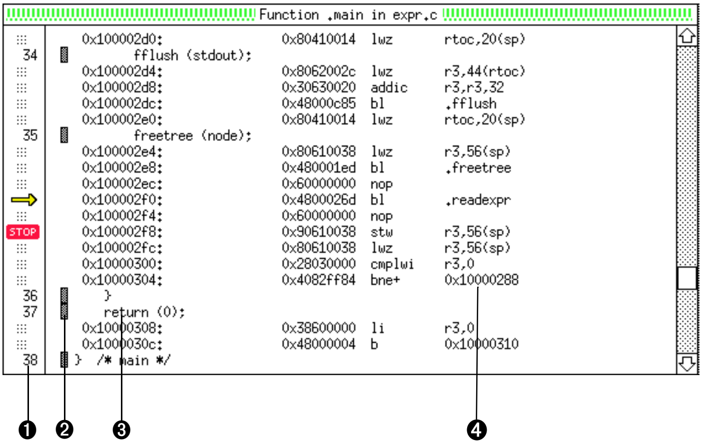

❶ Gridget (dotted grid) indicates action point can be set on an instruction
❷ Location by absolute address
❸ References by absolute address

FIGURE 49: **Address Only (Absolute Addresses)**




❶ Location by function and offsets
❷ References by function and offsets

FIGURE 50: **Assembler Only (Symbolic Addresses)**




❶ Source line
❷ Source line graphic indicator
❸ Location by absolute address
❹ References by absolute address

FIGURE 51: **Interleaved Source/Assembler (Absolute Addresses)**

# Current Stack Frame

You can return to the executing line of code for the current stack frame by selecting the **Current Stackframe** command from the **Current/Update/ Relatives** menu in the Process Window. This command forces the PC arrow onto the screen and discards the dive stack.

The **Current Stackframe** command is also useful if you want to undo the effect of scrolling or finding a function or file using the **Function or File...** command. For details, see "*Finding the Source Code for Functions*" on page 115.

If the program has not begun to run, the **Current Stackframe** command puts you in the first executable line of code in your main program function or subroutine.

# Editing Source Text

You can use the **Edit Source Text** command on the **Display/Directory/ Edit** menu to edit source files while you are debugging. TotalView starts your editor on the source file being displayed in the Source Code Pane of the Process Window.

## Changing the Editor Launch String

TotalView uses the editor launch string to determine how to start your editor. To change the value of the editor launch string, see "*Changing the Editor Launch String*" on page 121.

You can change the editor launch string to control the way TotalView starts your editor when you use the **Edit Source Text** command.

TotalView expands the editor launch string into a command string that is then executed by the shell **sh**. This allows you to configure exactly how the editor is started.

TotalView recognizes replacement characters in the launch string, which are expanded before TotalView starts your editor. The items that are expanded are as follows:

%E Expands to the value of the EDITOR environment variable, or to **vi** if EDITOR if not set.

%N Expands to the line number in the middle of the Source Code Pane. Use this option if your editor allows you to specify an initial line number at which to position the cursor.

%S Expands to the source file name displayed in the Source Code Pane.

%F Expands to the font name used when you started TotalView.

The default editor launch string is:

```
xterm -e %E +%N %S
```

This creates an xterm window in which to run the editor. If you use an editor that creates its own X window, such as **emacs** or **xedit**, you do not need to create an xterm window, and you should change the editor launch string.

You can change the editor launch string using one of the following methods:

- Using an X resource. Refer to "TOTALVIEW*EDITORLAUNCHSTRING" on page 281 for more information.
- Using the **Editor Launch String** command contained on the **Display/Directory/Edit** menu of the Process Window.

# Interpreting Status and Control Registers

The Stack Frame Pane in the Process Window lists the contents of CPU registers for the selected frame (you may need to scroll past the stack local variables to see them). To learn about the meaning of these registers, you need to consult the user's guide for your CPU and Appendix C, "*Architectures*" on page 343.

For your convenience, TotalView displays the bit settings of certain CPU registers symbolically, such as the registers that control the rounding and exception enable modes. You can edit the values of these registers and continue execution of your program. For example, you might do this to examine the behavior of your program with a different rounding mode.

Since the registers that are displayed vary from platform to platform, see Appendix C, "*Architectures*" on page 343 for information on the registers supported for your CPU. For general information on editing the value of variables (including registers), refer to "*Displaying Areas of Memory*" on page 146.

# Stopping Processes and Threads

To stop a process or a thread, go to the Process Window and select one of the following commands from the **Go/Halt/Step/Next/Hold** menu.

TABLE 12: Stopping a Process

| Command | Accelerator | Stops the ... |
|---|---|---|
| **Halt Process** | h | Process |
| **Halt Thread** | ^H | Thread; this is disabled if asynchronous thread control is not available |

TABLE 12: Stopping a Process (cont.)

| Command | Accelerator | Stops the ... |
|---|---|---|
| **Halt Group** | H | Process and all related processes. Issuing **Halt Group** on a process that is already stopped stops the other members of the program group |

When you stop a process, TotalView updates the Process Window and all related windows. When you restart the process, execution continues from the point where the process stopped.

You can force the Process Window to update process information by using the **Update Process Info** command from the **Current/Update/Relatives** menu without stopping the process. TotalView will temporarily stop the process so that it can reread the thread registers and memory. This allows you to quickly refresh your view of a process.

# Holding and Releasing Processes

TotalView allows you to hold and release processes. When a process is held, any command that tells the process to run, such as **Go Process** or **Go Group**, has no effect.

Manual hold and release are useful in a number of cases:

- If you wish to run a subset of the processes, you can manually hold all but the ones you want to run.
- If a process is held at a process barrier point and you want to run it without first running all the other processes in the group to that barrier, you can release it manually and then run it.

A process may also be held if it stops at a process barrier breakpoint. You can manually release a process being held at a process barrier breakpoint. See "*Process Barrier Breakpoints*" on page 206 for more information on manually holding and releasing process barrier breakpoint.

When a process is being held, the Root Window and Process Window display a held indicator. (This is a letter **H**.)

Here are the three ways to hold or release a process or group of processes:

- You can hold a group by choosing **Hold Group** command from the **Go/Halt/Step/Next/Hold** menu in the Process Window.
- You can then release the group by choosing **Release Group** command from the **Go/Halt/Step/Next/Hold** menu in the Process Window.
- You can toggle the hold/release state of a process by choosing the **Hold/Release Process** command from the **Go/Halt/Stop/Next/Hold** menu in the Process Window.

If a process or a thread is running when you issue a hold or release command, TotalView first stops the process or thread, then holds it.

**NOTE** Releasing a process does not mean that the thread will resume executing; execution only resumes after you use one of the stepping commands.

# Examining Process Groups

When you debug a multiprocess program, TotalView adds each process to two process groups as the process starts.

**NOTE** These groups are not related to UNIX process groups or PVM groups.

TotalView groups the processes depending on the type of system call (**fork()** or **execve()**) that created or changed the processes. The two types of process groups are:

**Program Group** Includes the parent process and all related processes. A program group includes children that were forked (processes that share the same source code as the parent) and children that were forked but which subsequently called by function's such as **execve()**. That is, these processes do not share the same source code as the parent.

Members of a program group can be stopped as a group.

**Share Group** Is the processes in a share group that share the same source code. Members of the same share group share action points.

In general, if you are debugging a multiprocess program, the program group is partitioned into more than one share group when the program has forked children that call **execve()**.

TotalView names processes based upon the name of the source program. Here are the naming rules TotalView uses:

- TotalView names the parent process after the source program.
- Child processes that are forked have the same name as the parent, but with a numerical suffix (.*n*).
- Child processes that call **execve()** after they are forked have the parent's name, the name of the new executable in angle brackets (<>) and a numerical suffix.

For example, if the **generate** process forks no children, and the **filter** process forks a child process that subsequently calls itself and then calls **execve()** to execute the **expr** program, TotalView names and groups the processes as shown in the following figure.

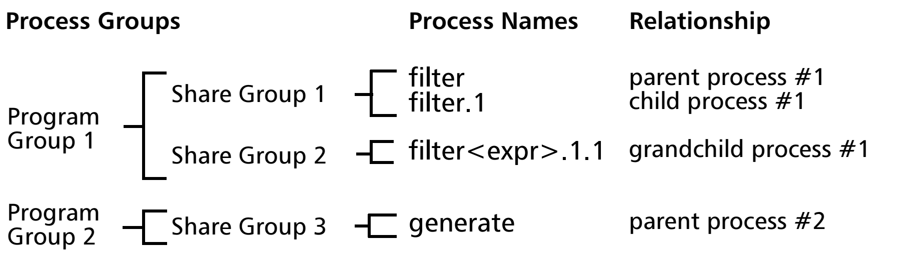
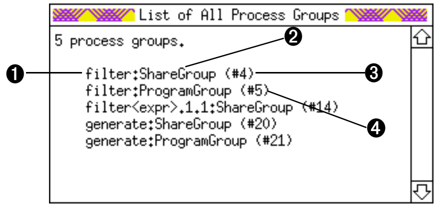



FIGURE 52: **Example of Program Groups and Share Groups**

## Displaying Process Groups

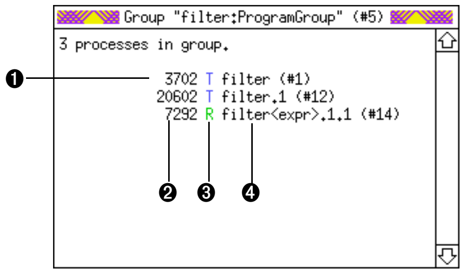
The Root Window displays the names of individual processes that are not in process groups. To display a list of process groups, select the **Show All Process Groups** command from the Root Window. The **Process Groups** Window appears, as shown in Figure 53.

If you dive into a process group listed in the window, a single Process Group Window appears, as shown in Figure 54. (You can also dive into any process listed in the Root Window to display its Process Window.)




❶ Name of executable
❷ Type of process group
❸ Group number
❹ Dive into process group to display single process group Window

FIGURE 53: **Process Groups Window**




❶ Dive into process to display Process Window
❷ Process ID
❸ State
❹ Process name

FIGURE 54: **Single Process Group Window**

## Changing Program Groups

In most situations, TotalView places a process in the correct program group. You can, however, move processes into different program groups. When you move a process into a different group, TotalView automatically places it in the associated share group. The advantage of moving a process into a different program group is that members of the same program group can start and stop on a breakpoint at the same time and be stepped as a group. (See "*Group-level Single Stepping*" on page 131 for details that apply to multiprocess programs.)

TotalView uses the name of the executable to determine the share group to which the program belongs. It does not examine the program to see if it is identical to another program with the same name; TotalView assumes the programs are identical because their names are identical.

TotalView does not expand a program's full pathname, so if one instance of a program is named with the full pathname (**./foo**), and another is named with the filename (**foo**), the programs are placed in different share groups.

To move a process into a different program group:

**1** Select **Show All Process Groups** from the Root Window. The Process Groups Window appears.

**2** Make note of the group ID number for the program group into which you will move the process. This number is displayed in parentheses.

**3** From the Process Window for the process to be moved, display the **Arguments/Create/Signal** menu, and select **Set Process Program Group**. A dialog box appears, as shown in the following figure.

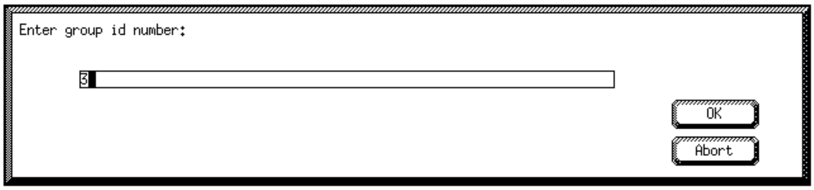



FIGURE 55: **Changing Process Groups Dialog Box**

**4** Enter the group ID number into the dialog box.

**5** Select **OK**.

## Finding Active Processes

Although a well-balanced multiprocess program distributes work evenly among processes, this situation does not always occur. If most active processes are waiting for work, it is tedious to look through the entire group to find the processes. Instead, you can use the **Find Interesting Relative** command to find them quickly.

After selecting the **Find Interesting Relative** command from the **Current/Update/Relatives** menu, TotalView displays:

- A Process Group Window listing the processes in decreasing order of interest.
- A Process Window for the most interesting process in the group (if it does not already have a Process Window open).

To see processes that are less *interesting*, reexecute the **Find Interesting Relative** command, or dive into the processes listed in the Process Group Window.

Here are a few of the criteria TotalView uses when it looks for something *interesting*:

- Running processes are more interesting than stopped processes.
- Threads at breakpoints are more interesting than threads stopped at arbitrary locations.
- Processes having threads with larger stacks are more interesting than processes having smaller stacks.

# Starting Processes and Threads

To start a process, go to the Process Window and select a commands from the **Go/Halt/Step/Next/Hold** menu. The commands are shown in Table 13.

TABLE 13: Starting a Process

| Command | Accelerator | Action |
|---|---|---|
| **Go Process** | g | Creates and starts this process. Resumes execution if the process is not being held, already exists and is stopped, or is at a breakpoint. Starting a process causes all threads in the process to resume execution. |
| **Go Thread** | ^g | Starts this thread. This command is disabled if asynchronous thread control is not available (see "*Thread-level Control*" on page 132). |

TABLE 13: Starting a Process (cont.)

| Command | Accelerator | Action |
|---|---|---|
| **Go Group** | G | Creates and starts this process and all other processes in the multiprocess program (program group). Resumes execution of this process and the execution of all processes in the program group if the process is not being held, already exists and is stopped, or at a breakpoint.<br>Issuing **Go Group** on a process that's already running starts the other members of the program group. |

For a single-process program, **Go Process** and **Go Group** are equivalent. For a single-threaded process, **Go Thread** and **Go Process** are equivalent.

Commands that contain the term G*roup* (for example, **Go Group**) refer to all members of the program group.

**NOTE** **If TotalView is holding a process, these commands will not start the process or its threads. See "*Holding and Releasing Processes*" on page 123.**

## Creating a Process without Starting it

The **Create Process (without starting it)** command creates a process and stops it before any of your program executes. If a program is linked with shared libraries, TotalView allows the dynamic loader to map into these libraries. Creating a process without starting it is useful if you need to:

- Create watchpoints or change global variables after a process is created, but before it runs.
- Debug C++ static constructor code.

## Creating a Process by Single-Stepping

The TotalView single-stepping commands allow you to create a process and run it to a certain point in your programs. The Process Window single-stepping commands in the **Go/Halt/Step/Next/Hold** menu are as shown in the following table.

TABLE 14: Creating a Process By Stepping

| Command | Accelerator | Creates the process and ... |
|---|---|---|
| **Step (source line)** | s | Runs it to the first line of the **main()** routine |
| **Next (source line)** | n | Runs it to the first line of the **main()** routine; this is the same as **Step (source line)** |
| **Step (instruction)** | i | Stops it before any of your program executes |
| **Next (instruction)** | x | Stops it before any of your program executes; this is the same as **Step (instruction)** |
| **Run (to selection)** | r | Runs it to the line or instruction selected in the Process Window |

If a group-level or thread-level stepping command creates a process, it behaves the same as a process-level command.

# Single Stepping

TotalView's single-stepping commands allow you to:

- Execute one source line or machine instruction at a time.
- Run to a selected line, which acts like a temporary breakpoint.
- Run until a function call returns.

Single-step commands are on the **Go/Halt/Step/Next/Hold** menu of the Process Window, and operate at process, group, or thread level. A level affects which threads within a process and processes within a group are allowed to run while the single-stepping command is executing.

In all cases, single-step commands operate on the *primary thread*, which is the selected thread in the current Process Window.

On all platforms except Compaq Alpha Linux, TotalView uses *smart* single stepping to speed up single stepping of one-line statements containing loops and conditions, such as Fortran 90 array assignment statements. S*mart* stepping occurs when TotalView realizes that it doesn't need to step through an instruction. For example, assume that you have the following statements:

```
integer iarray (1000,1000,1000)
iarray = 0
```

These two statements cause one billion scalar assignments. If you machine step every instruction, you'll probably never get past this statement. S*mart* stepping means that TotalView will single step through the assignment statement at very close to your machine's speed.

## Process-level Single Stepping

The process-level single-step commands step the primary thread within the process and allow other threads in the process to run. Threads that reach the stopping point in advance of the primary thread continue executing. The primary thread must reach the stopping point before execution stops.

## Group-level Single Stepping

The group-level single-step commands step threads of a "step group" drawn from a share group and allow other threads in the program group to run. (Program and share groups described on page 124.) When you issue the command, TotalView identifies a thread within each process that is similar to the primary thread. These threads form a step group; TotalView steps this group and stops only when all its members come to the command stopping point. Similar processes are in the same share group (they execute the same code) and have at least one thread with a PC that matches the PC of the primary thread. When several threads in a process are similar to the primary thread, TotalView arbitrarily assigns one thread to the step group.

Membership in a step group can change while a group single-step command executes. A thread can leave the step group if its PC diverges from that of the primary thread, for example if it executes a conditional branch that moves away from the primary thread. A thread that is not included in the step group at command onset can synchronize execution with the primary thread. TotalView then includes these cases in the step group.

The **Run (to selection) Group** command does not work like the other group single-step commands. It stops when the primary thread and at least one thread from each process in the share group reach the command stopping point. This lets you synchronize a group of processes and bring them to one location.

## Thread-level Single Stepping

The thread-level single-step commands step the primary thread to the command stopping point while holding other user threads in the process stopped. If, however, TotalView can identify manager threads, it runs them as it steps the single thread. Otherwise, TotalView runs the primary thread by itself.

**NOTE** Some operating systems only implement a synchronous run model: when one thread in the process runs, all threads must run. To step a thread on these systems, you must use the full-process, single step commands. These platforms include IRIX and SunOS.

Be aware that the thread-level single-step operations can fail to complete if the primary thread needs to synchronize with a thread that is not running. For example, if the primary thread requires a lock that another held thread owns, and steps over a call that tries to acquire the lock, the primary thread cannot continue successfully. The other thread must be allowed to run in order to release the lock.

## Thread-level Control

Only some operating systems allow a single thread to start and stop independently of others in the same process (this is known as asynchronous thread control). TotalView single thread commands are only available on the Compaq Tru64 UNIX, HP, and IBM AIX operating systems.

### Selecting Source Lines

Several of the single-stepping commands require you to select a source line or machine instruction in the Source Code Pane. To choose a source line, place the cursor over the line and select it. To deselect a source line, select it again. See "*Displaying Thread and Process Locations*" on page 138 for information on what occurs within the Root Window when you select a source line or machine instruction.

If you select a source line that has more than one instantiation (for example, in a C++ function template or code in a header file), TotalView displays a dialog box that allows you to select a specific instantiation, as shown in the following figure.

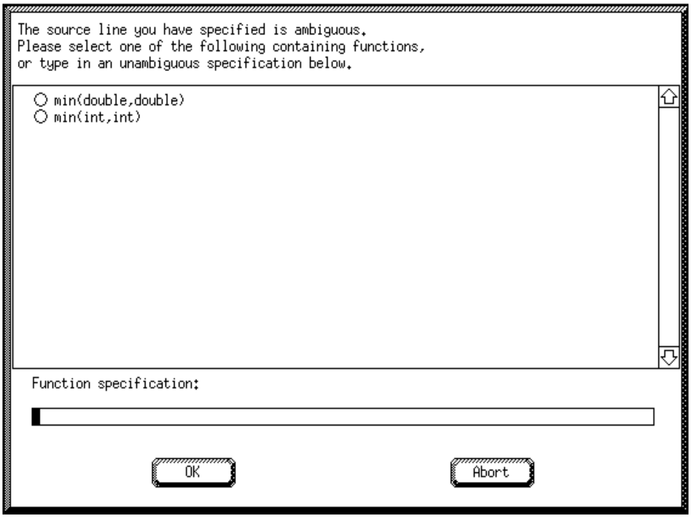



FIGURE 56: **Resolving Ambiguous Source Line Dialog Box**

You can now select a function or type in the function specification.

## Single-Step Commands

To execute a single-step command, select a thread and then use the **Go/Halt/Step/Next/Hold** menu in the Process Window to select a command.

The following applies to all single step command:

- To cancel a single-step command in progress, position the mouse pointer in the Process Window and press Ctrl-C.
- If your program reaches a breakpoint while stepping over a function, TotalView cancels the operation and your program stops at the breakpoint.
- If you issue a source line step command and the primary thread is executing in a function that has no source line information, TotalView performs the corresponding instruction step instead.

## Stepping Into Function Calls

The stepping functions execute a single source line or instruction. If the source line or instruction names a function, TotalView steps into it. If the source does not exist, TotalView displays the machine instructions for the function.

The source line stepping commands are shown in the following table.

TABLE 15: Source Line Stepping Commands

| Command | Accelerator | Executes a single source line at |
|---|---|---|
| **Step (source line)** | s | Process-level |
| **Step (source line) Group** | S | Group-level |
| **Step (source line) Thread** | M-^s | Thread-level |

The machine instruction stepping commands are described in the following table.

TABLE 16: Machine Stepping Commands

| Command | Accelerator | Executes a single machine instruction at |
|---|---|---|
| **Step (instruction)** | i | Process-level |
| **Step (instruction) Group** | I | Group-level |
| **Step (instruction) Thread** | M-^i | Thread-level |

The next section describes commands that allow you to single-step over a function call.

### Stepping Over Function Calls

When you step over a function, TotalView stops execution when the primary thread returns from the function and reaches the source line or instruction after the function call.

The commands that execute a single source line while stepping over functions are shown in the following table.

TABLE 17: Source Line Stepping Commands

| Command | Accelerator | Executes a single source line at |
|---|---|---|
| **Next (source line)** | n | Process-level |
| **Next (source line) Group** | N | Group-level |
| **Next (source line) Thread** | M-^n | Thread-level |

The commands that execute a single machine instruction while stepping over functions are shown in the following table.

TABLE 18: Machine Stepping Commands

| Command | Accelerator | Executes a single machine instruction at |
|---|---|---|
| **Next (instruction)** | x | Process-level |
| **Next (instruction) Group** | X | Group-level |
| **Next (instruction)Thread** | M-^x | Thread-level |

## Executing to a Selected Line

You don’t have to set a breakpoint to stop execution on a specific line because TotalView lets you run your program to a selected line or machine instruction. After selecting the line on which you want the program to stop, invoke one of the commands shown in the following table.

TABLE 19: Run to Selection Commands

| Command | Accelerator | Runs the ... |
|---|---|---|
| **Run (to selection)** | r | Process until the primary thread reaches the selected line. |

TABLE 19: Run to Selection Commands (cont.)

| Command | Accelerator | Runs the ... |
|---|---|---|
| **Run (to selection) Thread** | M-^r | Primary thread (which can be a manager thread) until it reaches the selected line. |
| **Run (to selection) Group** | R | Primary thread and all processes in the share group until it and at least one thread from each process in the share group reach the selected line. This command allows you to synchronize a group of processes and bring them to one location. |

If your program reaches a breakpoint while running to a selected line, the debugger discards the "run to" operation and stops at the breakpoint.

You can also run to a selected line in a nested stack frame, as follows:

**1** Select a nested frame in the Stack Trace Pane.
**2** Select a source line or instruction within the function.
**3** Issue a **Run (to selection)** command.

TotalView executes the primary thread until it reaches the selected line in the selected stack frame.

If your program calls recursive functions, you can select a nested stack frame in the Stack Trace Pane to tailor execution even more. In this situation, TotalView uses the frame pointer (FP) of the selected stack frame and the selected source line or instruction to determine when to stop execution. When your program reaches the selected line, TotalView compares the value of the selected FP to the value of the current FP:

- If the value of the current FP is deeper (more deeply nested) than the value of the selected FP, TotalView automatically continues your program.
- If the value of the current FP is equal or shallower (less deeply nested) than the value of the selected FP, TotalView stops your program.

## Executing to the Completion of a Function

You can step your program out of a function call. To finish executing the current function in a thread, select one of the commands shown in Table 20.

TABLE 20: Run to Selection Commands

| Command | Accelerator | Runs the ... |
|---|---|---|
| **Return (out of function)** | o | Process until the primary thread returns from the current function. |
| **Return (out of function) Group** | O | Primary thread and all the processes in the share group until the primary thread returns form the current function. |
| **Run (to selection) Thread** | M-^o | Primary thread (which can be a manager thread) until it returns from the current function. |

When the command completes, the primary thread is left stopped at the instruction after the one that called the function.

You can also return out of several functions at once, by selecting a nested stack frame in the Stack Trace Pane and then issuing a **Return (out of function)** command.

TotalView executes the primary thread until it returns to the function in the selected frame.

If your program calls recursive functions or mutually recursive functions, you can select a nested stack frame in the Stack Trace Pane to tailor completion of the function even more. In this situation, TotalView uses the frame pointer (FP) of the selected stack frame and the selected source line or instruction to determine when to stop execution. When your program reaches the selected line, TotalView compares the value of the selected FP with the value of the current FP in the following way:

- If the value of the current FP is deeper (more deeply nested) than the value of the selected FP, TotalView continues executing your program.
- If the value of the current FP is equal or shallower (less deeply nested) than the value of the selected FP, TotalView stops your program.

# Displaying Thread and Process Locations

You can see which processes and threads in the share group are at a location by selecting a source line or machine instruction in the Source Code Pane of the Process Window. TotalView dims thread and process information in the Root Window for share group members if the thread or process is not at the selected line. A process is considered at the selected line if any of the threads in the process are at that line. Selecting a line in the Process Window that is already selected, removes the dimming in the Root Window.

The Root Window reflects the line that you selected most recently. If you have several Process Windows open, the display in the Root Window will change depending on the line you selected last in a Process Window. The display can also change after an operation that changes the process state or when you issue an **Update Process Info** command.

The following figure shows Root Windows with dimmed process information and the corresponding Process Windows that create this output. In this example, the parallel program was run to a barrier breakpoint, and one process (**mpirun<cpi>.0**) was single-stepped to the next source line. In the top half of the figure, the line of source at the barrier breakpoint in the Process Window was selected. The Root Window shows the processes at that line not dimmed, and one process not at that line dimmed.

In the bottom half of the figure, the line at which the process stopped was selected. This process (**mpirun<cpi>.0**) is not dimmed, but the others are. Finally, since the MPI starter process (**mpirun**) is not in the same share group as the processes running the **cpi** program, the process information is not dimmed.

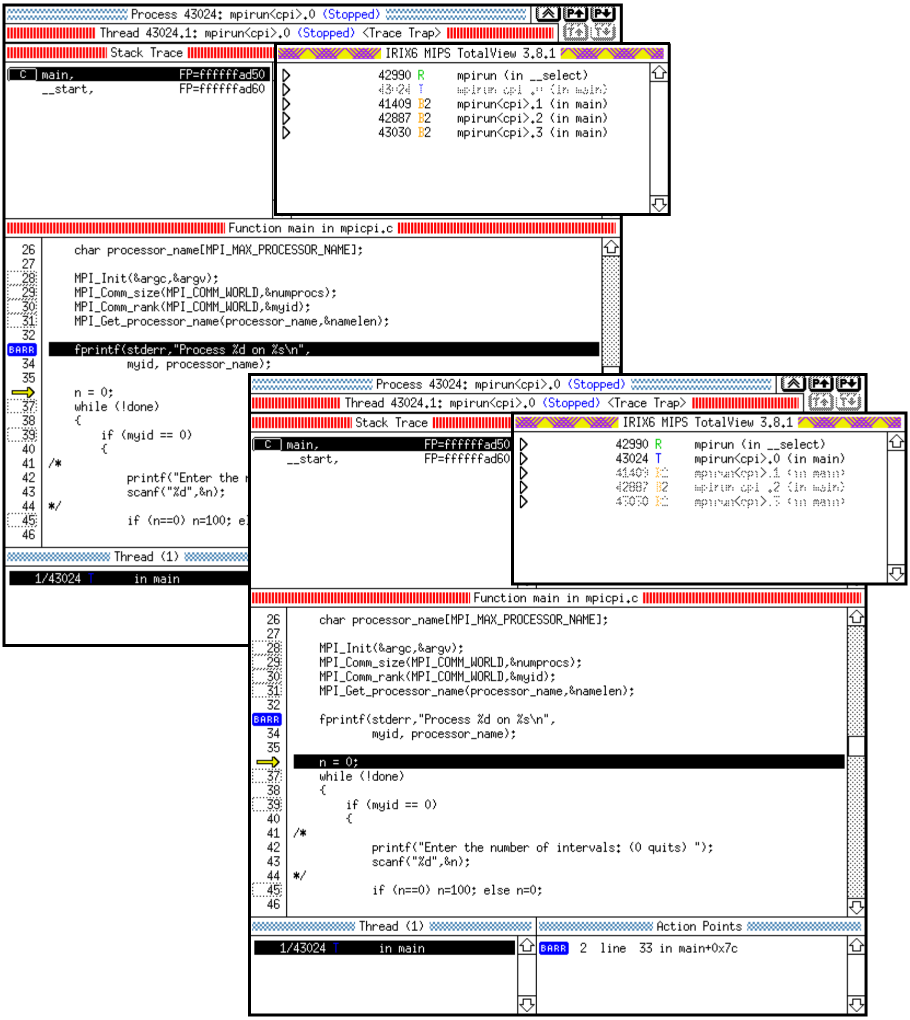


FIGURE 57: **Dimmed Process Information in the Root Window**

# Continuing with a Specific Signal

Letting your program continue to execute with a specific signal is useful when your program contains a signal handler. Here's how you tell TotalView that this should occur:

1. Display the **Go/Halt/Step/Next/Hold** menu and select the **Set Continuation Signal** command.
2. In the dialog box, enter the name (such as SIGINT) or number (such as 2) of the signal to be sent to the thread.
3. Select **OK**.

   The continuation signal is set for the thread you are focused on in the Process Window. If the operating system can deliver multithreaded signals, you may set a separate continuation signal for each thread. If it cannot, this command clears any continuation signal specified for other threads in the process.
4. Continue execution of your program with commands such as **Go**, **Step**, **Next,** or **Detach from Process**.

   TotalView continues the threads with the specified signals.

# Setting the Program Counter

You might find it useful to resume the execution of a thread at some statement other than the one where it stopped. You can do this by resetting the value of the program counter (PC). For example, you might want to skip over some code, execute some code again after changing certain variables, or restart a thread that is in an error state.

Setting the program counter can be crucial when you want to restart a thread that is in an error state. Although the PC icon in the tag field points to the source statement that caused the error, the PC actually points to the failed machine instruction within the source statement. You need to explicitly reset the PC to the correct instruction. (You can verify the actual location of the PC before and after resetting it by displaying it in the Stack Frame Pane or displaying interleaved source and assembler code in the Source Code Pane.)

In TotalView, you can set the PC of a stopped thread to a selected source line, a selected instruction, or an absolute value (in hexadecimal). When you set the PC to a selected line, the PC points to the memory location where the statement begins. For most situations, setting the PC to a selected line of source code is all you need to do.

To set the PC to a selected line:

1. If you need to set the PC to a location somewhere within a line of source code, display the **Display/Directory/Edit** menu and select the **Interleave Display Mode)** command. TotalView responds by displaying the assembler code.
2. Select the source line or instruction in the Source Code Pane. The debugger highlights the line in reverse video.

   If you select a line in a C++ function template that has more than one instantiation, TotalView asks you to select an instantiation. See "*Selecting Source Lines*" on page 133 for a description of how this works.
3. Display the **Go/Halt/Step/Next/Hold** menu and select the **Set PC to Selection...** command. TotalView asks for confirmation, resets the PC, and moves the PC icon to the selected line.

When you select a line and ask TotalView to set the PC to that line, TotalView attempts to force the thread to continue execution at that statement in the currently selected stack frame. If the currently selected stack frame is not the top stack frame, TotalView asks if it can unwind the stack:

```
This frame is buried. Should we attempt to unwind the stack?
```

If you select **Yes**, TotalView discards deeper stack frames (that is, all stack frames that are more deeply nested than the selected stack frame) and resets the machine registers to their values for the selected frame. If you select **No**, TotalView sets the PC to the selected line, but it leaves the stack and registers in their current state. Since you cannot assume that the stack and registers have correct values, selecting **No** is not usually the right thing to do.

**NOTE** In general, only advanced users should set the PC to an absolute address as it is extremely easy to crash your program. If you need to do this, make sure you have the correct address; no verification is done.

To set the PC to an absolute address:

1. Display the **Go/Halt/Step/Next/Hold** menu and select the **Set PC to Absolute Value...** command. TotalView than asks you to enter a hexadecimal address.
2. Enter the hexadecimal address into the dialog box.

**3** Select **OK**. TotalView resets the PC and moves the PC arrow to the line containing the absolute address.

# Deleting Programs

To delete all the processes in a program group, display the **Arguments/Create/Signal** menu and select the **Delete Program** command. The next time you start the program, for example, by using the **Go Process** command, TotalView creates and starts a fresh master process.

# Restarting Programs

You can use the **Restart Program** command to restart a program that is running or one that is stopped but has not exited. To restart a program, choose **Restart Program** from the **Arguments/Create/Signal** menu in the Process Window.

If the process is part of a multi-process program, TotalView deletes all related processes, restarts the master process, and runs the newly created program.

The **Restart Program** command is equivalent to the **Delete Program** command followed by the **Go Process** command.

Chapter 7

# Examining and Changing Data

This chapter explains how to examine and change data as you debug your program. You'll learn how to:

- Display Variable Windows
- Dive into variables
- Change the values of variables
- Change the data types of variables
- Change the addresses of variables
- Display machine instructions
- Display C++ and Fortran types
- Display array slices
- Filter and sort array data
- Display the value of a variable in all processes or threads
- Visualize array data
- Display threads objects

## Displaying Variable Windows

You can create windows that display local variables, registers, global variables, areas of memory, and machine instructions.

### Displaying Local Variables and Registers

In the Stack Frame Pane of the Process Window, you can dive into a formal parameter, local variable, or register to display a Variable Window. You can also dive into formal parameters and local variables in the Source Code

Pane. The Variable Window lists the name, address, data type, and value for the object, as shown in the following figure.

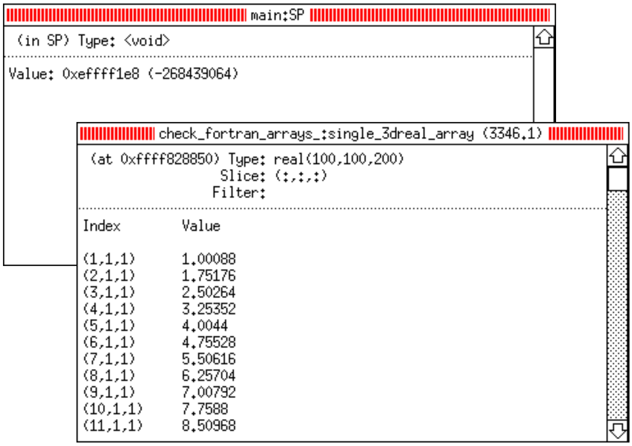



FIGURE 58: **Diving into Local Variables and Registers**

The top window is for a register while the bottom window is for a local array variable.

You can also display a local variable using the **Variable...** command that is contained on the **Function/File/Variable** menu of the Process Window. When prompted, enter the name of the variable in the dialog box.

If you keep Variable Windows open while you run a process or thread, the debugger updates the information in the windows when the process or thread stops. If TotalView cannot find a stack frame for a displayed local variable, it displays **Stale** in the pane header to warn you that you cannot trust the data, since no such variable exists.

When you debug recursive code, TotalView does not automatically refocus a Data Pane onto different invocation of a recursive function. If you have a breakpoint in a recursive function, you may need to explicitly open a new Data Pane to see the value of a local variable for that stack frame.

## Displaying a Global Variable

You can display a global variable by:

- Diving into the variable in the Source Code Pane.
- Displaying the **Function/File/Variable** menu and selecting the **Variable...** command. When prompted, enter the name of the variable.

A Variable Window appears for the global variable, as shown in the following figure.

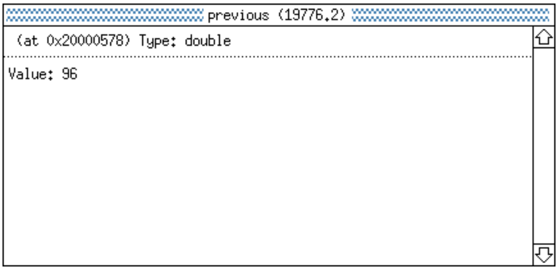



FIGURE 59: **Variable Window for a Global Variable**

## Displaying All Global Variables

TotalView lets you display all of the current process's global variables by selecting the **Global Variables Window** command from the **Function/File/Variable** menu. The window that appears contains the name and value of every global variable used by the process, as shown in the following figure.

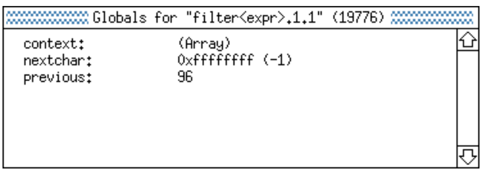



FIGURE 60: **Global Variables Window**

You can display a Variable Window for any global variable listed in this window by diving into the variable or by selecting the **Variable** command and entering a variable's name in the displayed dialog box.

## Displaying Areas of Memory

You can display areas of memory in hexadecimal and decimal. Do this by displaying the **Function/File/Variable** menu and selecting the **Variable** command. When prompted, enter one of the following in the dialog box:

- **A hexadecimal address**

  When you enter a single address, TotalView displays the word of data stored at that address.

- **A pair of hexadecimal addresses**

  When you enter a pair of addresses, TotalView displays the data (in word increments) from the first to the last address. To enter a pair of addresses, enter the first address, a comma, and the last address.

  NOTE  All hexadecimal constants must have a "0x" prefix. Also, you can enter these addresses using expressions.

The Variable Window for an area of memory, shown in the following figure, displays the address and contents of each word.

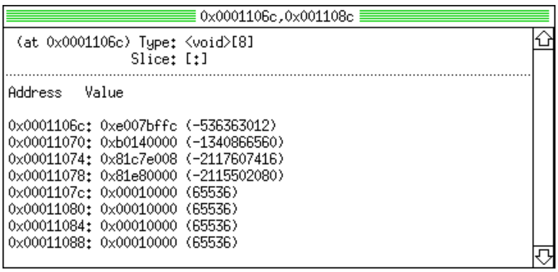



FIGURE 61: **Variable Window for Area of Memory**

The starting location of the memory area is displayed in the window's title. Within the window, information is displayed in hexadecimal and in decimal.

## Displaying Machine Instructions

You can display the machine instructions for entire routines as follows:

- Dive into the address of an assembler instruction in the Source Code Pane (such as **main+0x10** or **0x60**). A Variable Window displays the instructions for the entire function and highlights the instruction that you dived into.
- Dive into the PC in the Stack Frame Pane. A Variable Window lists the instructions for the entire function containing the PC, and highlights the instruction to which the PC points.

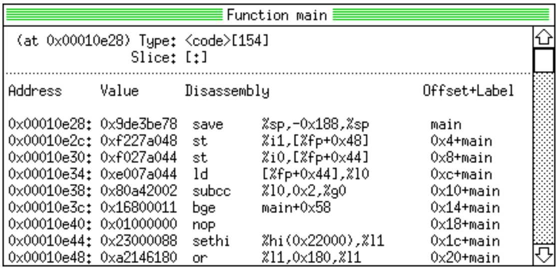



FIGURE 62: **Variable Window with Machine Instructions**

- Cast a variable to type **<code>** or array of **<code>**, as described in "*Changing Types to Display Machine Instructions*" on page 158.

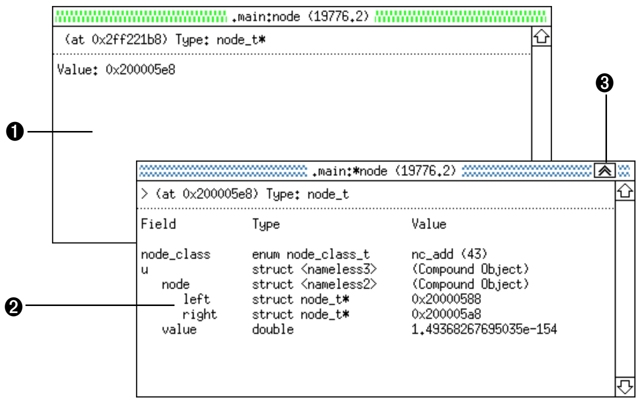
## Closing Variable Windows

When you are finished analyzing the information in a Variable Window, use the **Close Window** command to close the window. You can also use the **Close All Similar Windows** command to close all Variable Windows.

# Diving in Variable Windows

If the variable being displayed in a Variable Window is a pointer, structure, or array, you can dive into the contents listed in the Variable Window. This additional dive is called a *nested dive*. When you perform a nested dive, the Variable Window replaces the original information with information about the current variable. With nested dives, the original Variable Window is known as the *base window*.

Figure 63 shows the results of diving into a variable in the Stack Frame Pane of **main()** in the Process Window. This example dives into a pointer variable named **node** with a type of **node_t***. The first Variable Window (the base window) displays the value of **node**.




❶ Base window: First dive (on the variable node_t*, a pointer)
❷ Nested window: Second dive (on the value of node_t
❸ Undive icon

FIGURE 63: **Nested Dives**

Diving into the value shown in the base window tells TotalView to replace the window with a nested dive window. The nested dive window—displayed at the bottom of the figure—shows the structure referenced by the **node_t*** pointer.

Also, notice that the number of right angle brackets (>) in the upper left hand corner indicates the number of nested dives that were performed in the window. TotalView maintains each dive on a dive stack.

You can manipulate Variable Windows and nested dive windows in the following ways:

- To "undive" from a nested dive, click the **Dive** mouse button on the undive icon, and the previous contents of the Variable Window appears.

- If you have performed several nested dives and want to create a new copy of the base window, select the **New Base Window** command from the Variable Window.
- If you dive into a variable that already has a Variable Window open, the Variable Window pops to the surface. If you want a duplicate Variable Window open, hold down the Shift key when you dive on the variable.
- If you select the **Duplicate Window** command from the Variable Window, a new Variable Window appears that is a duplicate of the current vAriable Window except that it has an empty dive stack.

# Changing the Values of Variables

You can change the value of any variable or the contents of any memory location displayed in a Variable Window by selecting the value and using the field editor to change the value as desired.

You can type an expression instead of a value. For example, you can enter 1024*1024. This expression can include logical operators.

You can also edit the value of variables directly from the Stack Frame Pane by selecting them. You cannot, however, change the value of bit fields directly, but you can use the expression window to assign a value to a bit field. See "*Evaluating Expressions*" on page 233 Similarly, you cannot directly change the value of fields in nested structures; instead, you must first dive into it.

# Changing the Data Type of Variables

The data type declared for the variable determines its format and size (amount of memory) in the Variable Window. For example, if you declare an **int** variable, the debugger displays the variable as an integer.

You can change the way data is displayed in the Variable Window by editing the data type. This is known as *casting*. TotalView assigns types to all data types, and in most cases, they are identical to their programming language counterparts.

- When displaying a C variable, TotalView types are identical to C type representations, except for pointers to arrays. By default, TotalView uses a simpler syntax for pointers to arrays.
- When displaying a Fortran variable, TotalView types are identical to Fortran type representations for most data types, including **INTEGER**, **REAL**, **DOUBLE PRECISION**, **COMPLEX**, **LOGICAL**, and **CHARACTER**.

You can use the field editor to change a type in a Variable Window. If the window contains a structure with a list of fields, you can edit the types of the fields listed in the window.

**NOTE** **When you edit a type, TotalView changes how it displays the variable in the current Variable Window, but other windows listing the variable remain the same.**

## How TotalView Displays C Data Types

TotalView's syntax for displaying data is identical to C cast syntax for all data types except pointers to arrays. Thus, you use C cast syntax for **int**, **unsigned**, **short**, **float**, **double**, **union**, and all named **struct** types.

You read TotalView types from right to left. For example, **<string>*[20]*** is a pointer to an array of 20 pointers to **<string>**.

Table 21 shows some common types.

TABLE 21: Common types

| Type String | Meaning |
|---|---|
| int | Integer |
| int* | Pointer to integer |
| int[10] | Array of 10 integers |
| <string> | Null-terminated character string |
| <string>** | Pointer to a pointer to a null-terminated character string |
| <string>*[20]* | Pointer to an array of 20 pointers to null-terminated strings |

The following sections discuss the more complex types.

## C Cast Syntax

You can also enter C cast syntax verbatim in the type field for any type. In addition, TotalView can display C cast syntax permanently if you set an X Window Resource. See "TOTALVIEW*CTYPESTRINGS" on page 280 for further information.

## Pointers to Arrays

Suppose you declared a variable **vbl** as a pointer to an array of 23 pointers to an array of 12 objects of type **mytype_t**. The C language declaration for this is:

```
mytype_t (*(*vbl)[23]) [12];
```

To cast **vbl** to the same type in your C program:

```
(mytype_t (*(*)[23])[12])vbl
```

The TotalView type syntax for **vbl** is:

```
mytype_t[12]*[23]*
```

## Arrays

Array type names can include a lower and upper bound separated by a colon.

By default, the lower bound for a C or C++ array is 0, and the lower bound for a Fortran array is 1. In the following example, an array of integers is declared in C and then in Fortran:

```
int a[10];
integer a(10)
```

In the C example, the elements of the array range from **a[0]** to **a[9]**, while in the Fortran example, the elements of the array range from **a(1)** to **a(10)**.

When the lower bound for an array dimension is the default for the language, TotalView displays only the extent (that is, the number of elements) of the dimension. Consider the following array declaration in Fortran:

```
integer a(1:7,1:8)
```

Since both dimensions of the array use the default lower bound for Fortran (1), TotalView displays the data type of the array using only the extent of each dimension, as follows:

```
integer(7,8)
```

If an array declaration does not use the default lower bound, TotalView displays both the lower bound and upper bound for each dimension of the array. For example, in Fortran, you would declare an array of integers with the first dimension ranging from –1 to 5 and the second dimension ranging from 2 to 10 is as follows:

```
integer a(-1:5,2:10)
```

TotalView displays the following data type for this Fortran array:

```
integer(-1:5,2:10)
```

When editing a dimension of an array, you can enter just the extent (if using the default lower bound) or both the lower and upper bounds separated by a colon.

If desired, you can display a subsection of an array, or filter a scalar array for values matching a filter expression. Refer to "*Displaying Array Slices*" on page 167 and "*Array Data Filtering*" on page 171 for further information.

## Typedefs

The debugger recognizes the names defined with **typedef**, but displays the definition of the type (that is, the base data type), rather than its name. For example:

```
typedef double *dptr_t;
dptr_t p_vbl;
```

The debugger displays the type for **p_vbl** as **double***, not as **dptr_t**.

## Structures

TotalView treats **struct** as a keyword. You can type **struct** as part of the type string, but it is optional. If you have a structure and another data type with the same name, however, you must include **struct** with the name of the structure so TotalView can distinguish between the two data types.

If you name a structure using **typedef**, the debugger uses the **typedef** name as the type string. Otherwise, the debugger uses the structure tag for the **struct**.

For example, consider the structure definition:

```
typedef struct mystruc_struct {
    int field_1;
    int field_2;
} mystruc_type;
```

TotalView displays **mystruc_type** as the type for **struct mystruc_struct**.

TotalView does not interpret the definition of structures in a type string. For example, it cannot use a definition such as **struct {int a; int b;}**.

## Unions

TotalView displays a union as it does a structure. Even though the fields of a union are overlaid in storage, TotalView displays them on separate lines in the Variable Window.

When TotalView displays some complex arrays and structures, it displays the compound object or array types in the Variable Window. Editing the compound object or array types could yield undesirable results. We do not recommend editing these types.

## Built-In Types

TotalView provides a number of predefined types. These types are enclosed in angle brackets to avoid conflict with types already defined in the language. You can use these built-in types anywhere a user-defined type can be used, such as in a cast expression. These types are also useful when debugging executables with no debugging symbol table information. The following table lists the built-in types.

TABLE 22: Built-in Types

| Type String | Language | Size | Meaning |
|---|---|---|---|
| <address> | C | void* | Void pointer (address) |
| <char> | C | char | Character |

TABLE 22: Built-in Types (cont.)

| Type String | Language | Size | Meaning |
|---|---|---|---|
| <character> | Fortran | character | Character |
| <code> | C | parcel | Machine instructions<br>A parcel is defined to be the number of bytes required to hold the shortest instruction for the target architecture |
| <complex*16> | Fortran | complex*16 | real*8-precision floating-point complex number.<br>**complex*16** types contain a real part and an imaginary part, which are both of type **real*8** |
| <complex*8> | Fortran | complex*8 | real*4-precision floating-point complex number.<br>**complex*8** types contain a real part and an imaginary part, which are both of type **real*4** |
| <complex> | Fortran | complex | Single-precision floating-point complex number.<br>**complex** types contain a real part and an imaginary part, which are both of type **real** |
| <double precision> | Fortran | double precision | Double-precision floating-point number |
| <double> | C | double | Double-precision floating-point number |
| <extended> | C | long double | Extended-precision floating-point number.<br>Extended-precision numbers must be supported by the target architecture |
| <float> | C | float | Single-precision floating-point number |
| <int> | C | int | Integer |
| <integer*1> | Fortran | integer*1 | One-byte integer |

TABLE 22: Built-in Types (cont.)

| Type String | Language | Size | Meaning |
|---|---|---|---|
| <integer*2> | Fortran | integer*2 | Two-byte integer |
| <integer*4> | Fortran | integer*4 | Four-byte integer |
| <integer*8> | Fortran | integer*8 | Eight-byte integer |
| <integer> | Fortran | integer | Integer |
| <logical*1> | Fortran | logical*1 | One-byte logical |
| <logical*2> | Fortran | logical*2 | Two-byte logical |
| <logical*4> | Fortran | logical*4 | Four-byte logical |
| <logical*8> | Fortran | logical*8 | Eight-byte logical |
| <logical> | Fortran | logical | Logical |
| <long long> | C | long long | Long long integer |
| <long> | C | long | Long integer |
| <real*16> | Fortran | real*16 | Sixteen-byte floating-point number |
| <real*4> | Fortran | real*4 | Four-byte floating-point number |
| <real*8> | Fortran | real*8 | Eight-byte floating-point number |
| <real> | Fortran | real | Single-precision floating-point number |
| <short> | C | short | Short integer |
| <string> | C | char | Array of characters |
| <void> | C | long | Area of memory |

The following sections give more detail about several of the built-in types.

## Character arrays (<string> Data Type)

If you declare a character array as **char vbl[*n*]**, the debugger automatically changes the type to **<string>[*n*]**; that is, a null-terminated, quoted string with a maximum length of *n*. Thus, by default, the array is displayed as a quoted string of *n* characters, terminated by a null character. Similarly, the debugger changes **char*** declarations to **<string>*** (a pointer to a null-terminated string).

Since most C character arrays represent strings, the **<string>** type can be very convenient. If, however, you intended the **char** data type to be a pointer to a single character or an array of characters, you can edit the **<string>** back to a **char** (or **char[n]**) to display the variable as you declared it.

### Areas of memory (<void> Data Type)

TotalView uses the **<void>** type for data of an unknown type, such as the data contained in registers or in an arbitrary block of memory. The **<void>** type is similar to the **int** in the C language.

If you dive into registers or display an area of memory, the debugger lists the contents as a **<void>** data type. Further, if you display an array of **<void>** variables, the index for each object in the array is the address, not an integer. This address can be useful when displaying large areas of memory.

If desired, you can change a **<void>** into another type. Similarly, you can change any type into a **<void>** to see the variable in decimal and hexadecimal.

### Instructions (<code> Data Type)

TotalView uses the **<code>** data type to display the contents of a location as machine instructions. Thus, to look at disassembled code stored at a location, dive on the location and change the type to **<code>**. To specify a block of locations, use **<code>[n]**, where **n** is the number of locations being displayed.

## Type Casting Examples

This section contains some common type casting examples.

### Example: Displaying the argv Array

Typically, **argv** is the second argument passed to **main(),** and it is either a **char **argv** or **char *argv[ ]**. Since these declarations are equivalent (a pointer to one or more pointers to characters), TotalView converts both to **<string>**** (a pointer to one or more pointers to null-terminated strings).

Suppose **argv** points to an array of 20 pointers to character strings. There is how you can edit its type to display an array of 20 pointers:

1 Select the type string for **argv**.
2 Edit the type string using the field editor commands. Change it to:
```
<string>*[20]*
```
3 To display the array, dive into the value field for **argv**.

### Example: Displaying Declared Arrays

TotalView displays arrays in the same way as it displays local and global variables. In the stack frame or Source Code Pane, dive into the declared array. A Variable Window displays the elements of the array.

### Example: Displaying Allocated Arrays

The C language uses pointers for dynamically allocated arrays. For example:

```
int *p = malloc(sizeof(int) * 20);
```

Because TotalView does not know that **p** actually points to an array of integers, here is how you would display the array:

1 Dive on the variable **p** of type **int***.
2 Change its type to **int[20]***.
3 Dive on the value of the pointer to display the array of 20 integers.

## Opaque Type Definitions

An opaque type is a data type that is not fully specified, hidden, or whose declaration is deferred. For example the following C declaration defines the data type for **p** as pointer to **struct foo**, which is not yet defined:

```
struct foo;
struct foo *p;
```

When TotalView encounters this kind of information, it indicates its data type by appending **<opaque>** to the declaration. For example:

```
struct foo <opaque>
```

If the type is actually defined in another module, deleting **<opaque>** from the data type tells TotalView to find the actual definition for the type.

On platforms where TotalView uses “lazy reading” of the symbol table, you must force TotalView to read the symbols from the module containing the full type definition of the opaque type. Use the **Function or File** command to force TotalView to read the symbols, as described in “*Finding the Source Code for Functions*” on page 115.

# Changing the Address of Variables

You can edit the address of a variable in a Variable Window. When you edit the address, the Variable Window shows the contents of the new location.

You can also enter an address expression, such as **0x10b8–0x80**.

# Changing Types to Display Machine Instructions

Here’s how you can display machine instructions in any Variable Window:

1 Select the type string at the top of the Variable Window.
2 Change the type string to be an array of **<code>** data types, where *n* indicates the number of instructions to be displayed. For example:

```
<code>[n]
```

The debugger displays the contents of the current variable, register, or area of memory as machine-level instructions.

The Variable Window (shown in Figure 62 on page 147) lists the following information about each machine instruction:

| | |
|---|---|
| **Address** | The machine address of the instruction. |
| **Value** | The hexadecimal value stored in the location. |
| **Disassembly** | The instruction and operands stored in the location. |
| **Offset+Label** | The symbolic address of the location as a hexadecimal offset from a routine name. |

You can also edit the value listed in the **value** field for each machine instruction.

# Displaying C++ Types

## Classes

TotalView displays C++ classes and accepts **class** as a keyword. When you debug C++, TotalView also accepts the *unadorned* name of a class, struct, union, or enum in the type field. TotalView displays nested classes that use inheritance, showing derivation by indentation.

**NOTE** Some C++ compilers do not output accessibility information. In these cases, the information is omitted from the display.

For example, the following figure displays an object of a **class c**:

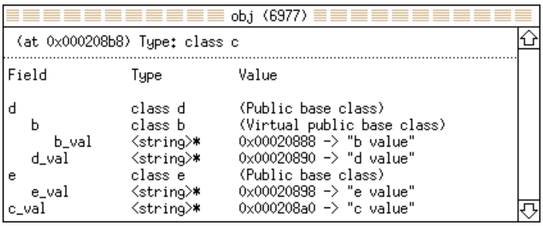



FIGURE 64: **Displaying C++ Classes that Use Inheritance**

The definition is as follows:

```
class b {
    char * b_val;
public:
    b() {b_val = “b value“;}
};

class d : virtual public b {
    char * d_val;
    public:
    d() {d_val = “d value“;}
};

class e {
    char * e_val;
 public:
    e() {e_val = “e value“;}
};
```

```
class c : public d, public e {
    char * c_val;
 public:
    c() {c_val = "c value";}
};
```

## Changing Class Types in C++

TotalView tries to display the correct data when you change the type of a Data Pane to move up or down the derivation hierarchy.

If a change in the data type also requires a change in the address of the data being displayed, the debugger asks you about changing the address. For example, if you edit the type field in **class c** shown in Figure 65 to **class e**, TotalView displays the following dialog box:

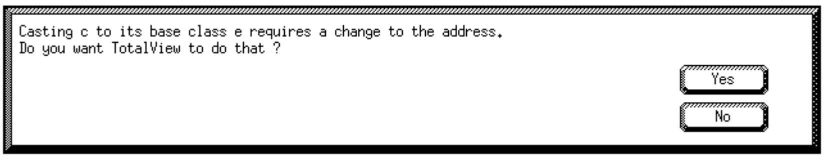



FIGURE 65: C++ Type Cast to Base Class Dialog Box

Selecting **Yes** tells TotalView to change the address to ensure that it displays the correct base class member. Selecting **No** tells TotalView to display the memory area as if it were an instance of the type to which it is being cast, leaving the address unchanged.

Similarly, if you change a data type in the Data Pane so you can cast a base class to a derived class, and that change requires an address change, the debugger asks you to confirm the operation. For example, the following figure show the dialog posted if you cast from **class e** to **class c**:

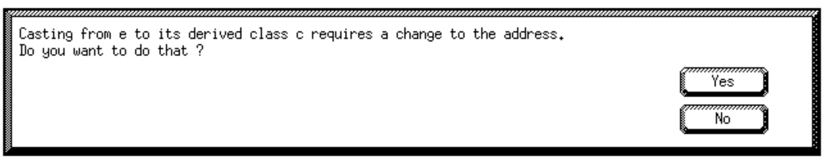



FIGURE 66: C++ Type Cast to Derived Class Dialog Box

# Displaying Fortran Types

TotalView allows you to display FORTRAN 77 and Fortran 90 data types.

## Displaying Fortran Common Blocks

For each common block defined within the scope of a subroutine or function, TotalView creates an entry in that function's common block list. The Stack Frame Pane in the Process Window displays the name of each common block for a function. The names of common block members have function scope, not global scope.

TotalView creates a user defined data type for the common block in which each of the common block members are fields in the type. If you dive on a common block name in the Stack Frame Pane, TotalView displays the entire common block in a Variable Window, as shown in Figure 67.

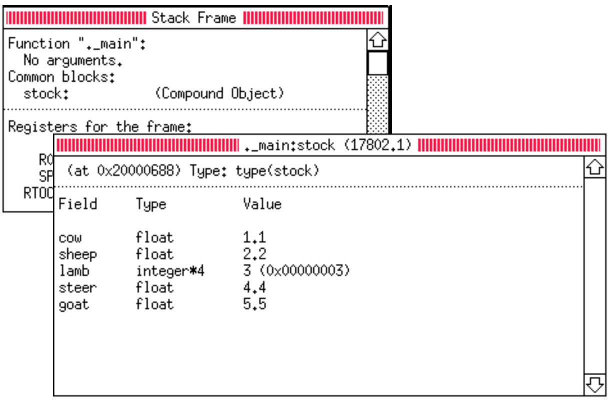



FIGURE 67: Diving into Common Block List in Stack Frame Pane

The top window shows a common block list in a Stack Frame Pane. The other window shows the results of diving on the common block to see its elements.

If you dive on a common block member name, TotalView searches all common blocks for a matching member name and displays the member in a Variable Window.

Normally, TotalView displays the initial address for a common block in the Data Pane. When the common block is a composite object with separate addresses for each component, TotalView displays a **Multiple** tag to indicate that it cannot display a single address.

## Displaying Fortran Module Data

TotalView tries to locate all data associated with a Fortran module and provide a single display that contains all of it. For functions and subroutines defined in a module, TotalView adds the full module data definition to the list of modules displayed in the Stack Frame Pane.

**NOTE** TotalView only displays a module if it contains data. Also, the amount of information that your compiler gives TotalView may restrict what is displayed.

Although a function may use a module, TotalView may not be able to determine if the module was used or what the true names of the variables in the module are. In this case, module variables either appear as local variables of the subroutine, or a module appears on the list of modules in the Stack Frame Pane that contains (with renaming) only the variables used by the subroutine.

Alternatively, you can view a list of all of the known modules by using the **Fortran Modules Window** command from the **Function/File/Variable** menu. This window behaves like the **Global Variables** Window, so you can dive through an entry to display the actual module data, as shown in Figure 68.

**NOTE** If you are using the SUNPro compiler, TotalView can only display module data if you force it to read the debug information for a file that contains the module definition or a module function. For more information, see "*Finding the Source Code for Functions*" on page 115.

## Debugging Fortran 90 Modules

Fortran 90 and 95 let you place functions, subroutines, and variables inside modules. These modules can then be **USE**d (included) elsewhere. When modules are **USE**d, the names in the module become available in the *using* compilation unit, unless they have been excluded by **USE ONLY**, or

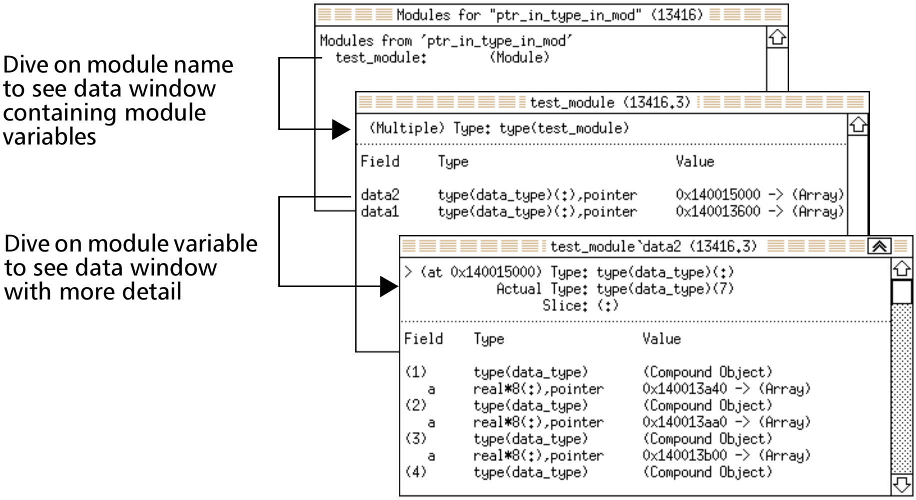



FIGURE 68: Fortran Modules Window

renamed. This means that you do not need to explicitly qualify the name of a module function or variable from the Fortran source code.

When debugging this kind of information, you will need to know the location of the function being called. Consequently, TotalView uses the following syntax when it displays a function contained within a module:

*modulename`functionname*

You can use also this syntax in the **Function or File** command in the **Function/File/Variable** menu.

Fortran 90 also introduced the idea of a contained function that is only visible in the scope of its parent and siblings. There can be many contained functions in a program, all using the same name. TotalView uses a similar syntax when it displays these function. If the compiler gave TotalView the function name for a nested function, TotalView displays it using the following syntax:

*parentfunction()`containedfunction*

If you give an ambiguous name for a function, TotalView displays a dialog box showing all of the possible matching functions. See "*Finding the Source Code for Functions*" on page 115 for more information.

Within contained functions, all of the parent function's variables are visible and accessible through a static chain. If the compiler retained information about the static chain, TotalView can access these variables in the same way as the compiled code does. Consequently, they are visible in Data Panes, and from evaluation points or expressions. If the compiler does not pass on information about the static chain, TotalView can still find these up-level variables and display then in Data Panes, but you will not be able to use them in evaluation points or expressions.

## Fortran 90 User Defined Type

A Fortran 90 user defined type is similar to a C structure. TotalView displays a user defined type as **type(***name***)**, which is the same syntax used in Fortran 90 to create a user defined type. For example, here's a code fragment that defines a variable **matrix1** of **type(***sparse***)**:

```
type sparse
logical*1, pointer :: smask (:,:)
real, pointer :: sval (:)
character (20) :: heading
end type sparse
type(sparse)::matrix1
```

TotalView displays it as follows:

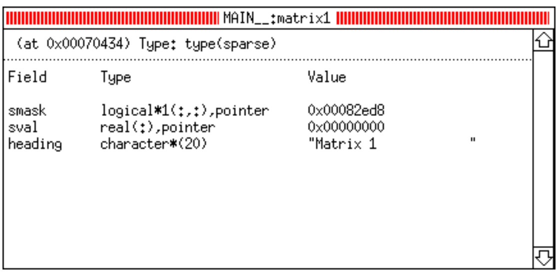



FIGURE 69: **Fortran 90 User Defined Type**

## Fortran 90 Deferred Shape Array Type

Fortran 90 allows you to define deferred shape arrays and pointers. The actual bounds of the array are not determined until the array is allocated, the pointer is assigned, or, in the case of an assumed shape argument to a subroutine, the subroutine is called. The type of deferred shape arrays is displayed by TotalView as *type***(:)**, in the same way that an array is declared in Fortran.

When TotalView displays the data for a deferred shape arrays, it displays the type used in the definition of the variable and the actual type that this instance of the variable has. The actual type is not editable since you can achieve the same effect by editing the definition's type. The following example shows the type of a deferred shape rank 2 array of **REAL** data with runtime lower bounds of -1 and 2, and upper bounds of 5 and 10:

```
       Type: real(:,:)
Actual Type: real(-1:5,2:10)
      Slice: (:,:)
```

## Fortran 90 Pointer Type

A Fortran 90 pointer type allows you to point to scalar or array types. The debugger displays pointer types as *type***,pointer**, which is the same syntax used in Fortran 90 to create a pointer variable.

For example, a **pointer** to a rank 1 deferred shape array of **real** data is displayed in the Variable Window as:

```
Type: real(:),pointer
```

To view the data instead of the pointer variable, you must dive on the value.

**NOTE** If you are using the IBM xlf compiler, TotalView cannot determine the rank of the array from the debugging information. In this case, the type of a pointer to an array appears as "type(...),pointer". The actual rank is filled in when you dive through the pointer to look at the data.

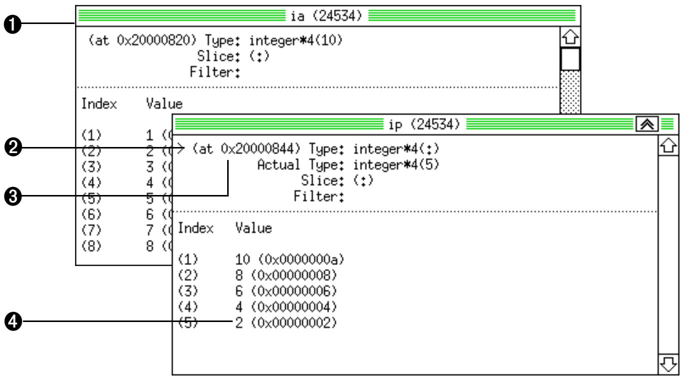
The value of the pointer is displayed as the address of the data to which the pointer points. This address is not necessarily the array element with the lowest address.

TotalView implicitly handles slicing operations that set up a pointer or assumed shape subroutine argument so that indices and values it displays in the **Variable** Window are the same as you would see in the Fortran code.

For example:

```
integer, dimension(10), target :: ia
integer, dimension(:), pointer :: ip
do i = 1,10
    ia(i) = i
end do
ip => ia(10:1:-2)
```

After diving through the **ip** pointer, TotalView displays the window shown in Figure 70.




❶ Target array ia
❷ Pointer ip into array ia
❸ Address of ip(1)
❹ Values reflect slice

FIGURE 70: Fortran 90 Pointer Value

Notice that the address displayed is not that of the array's base. Since the array's stride is negative, succeeding array elements are at lower absolute addresses. Consequently, the address displayed is that of the array element having the lowest index (which may not be the first displayed element if you used a slice to display the array with reversed indices).

# Arrays

TotalView can quickly display very large arrays in Variable Windows. If an array overlaps nonexistent memory, the initial portion of the array is correctly formatted. If memory is not allocated for an array element, TV displays "Bad Address" in the element's subscript.

## Displaying Array Slices

TotalView lets you display array subsections by editing the *slice field* within an array's Variable Window. (An array subsection is called a *slice*.) The slice field contains placeholders for all array dimensions. For example, here is a C declaration for a three-dimensional array:

```
integer ia[10][20][5]
```

TotalView defines this slice as follows: [:][:][:].

Here is a Fortran 90 deferred shape array definition:

```
integer, dimension (:,:) :: ia
```

Its TotalView slice definition is **(:,:)**.

As you can see, TotalView displays as many colons (:) as there are array dimensions. Initially, the field contains **[:]** for C arrays or **(:)** for Fortran arrays.

### Slice Definitions

A slice definition has the following form:

*lower_bound*:*upper_bound*:*stride*

This tells TotalView to display every *stride* element of the array, starting at the *lower_bound* and continuing through the *upper_bound*, inclusive. (A *stride* tells TotalView that it should skip over elements and not display them.)

For example, if you specified a slice of **[0:9:9]** for a 10-element C array, TotalView displays the first element and last element, which is the 9th element beyond the lower bound.

If a slice is defined as **[lb:ub:stride]**, Totalview represents the set of values of **i** generated by the **append** statements in the following way:

```
i = lb
if (stride > 0)
    while ( i <= ub )
          append i
          i = i + stride
```

```
else
    while ( i >= ub )
          append i
          i = i + stride
```

If **stride < 0** and **ub > lb**, TotalView treats the slice as if it were as follows:

```
[ub : lb : stride]
```

(This is an extension to the way Fortran displays slices.) Consequently, the debugger lets you view a dimension with reversed indexing. For example, the following definition tells TotalView to display an array beginning at its last value and moving to its first:

```
[::-1]
```

In contrast, Fortran 90 requires that you explicitly enter the upper and lower bounds when you are reversing the order in which array elements are displayed.

Because the default value for the stride is **1**, you can omit the stride (and the colon that precedes it) if your stride value is 1. For example, the following two definitions display array elements 0 through 9:

```
[0:9:1]
[0:9]
```

If the lower and upper bound are the same number, you can specify the slice with just a single number. This number indicates the lower and upper bound. For example, the following two definitions tell TotalView to display array element 9:

```
[9:9:1]
[9]
```

**NOTE** **The lower_bound, upper_bound, and stride can only be constants.**

For multidimensional arrays, you can specify a slice for each dimension using the following syntax:

| | |
|---|---|
| C and C++: | [*slice*][*slice*]... |
| Fortran: | (*slice*,*slice*,...) |

## Example 1

A slice declaration of **[::2]** for a C or C++ array (with a default lower bound of 0) tells TotalView to display elements with even indices of the array: 0, 2, 4, and so on. However, if this were defined for a Fortran array (with a default lower bound of 1), TotalView displays elements with odd indices of the array: 1, 3, 5, and so on.

## Example 2

The following example displays a slice of **(::9,::9). This definition** displays the four corners of a 10-element by 10-element Fortran array.

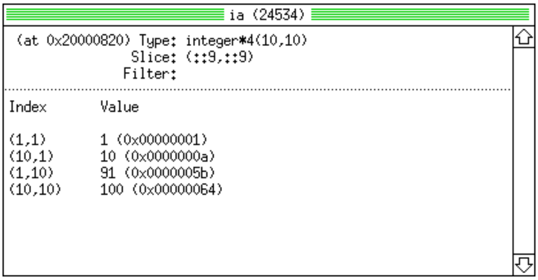



FIGURE 71: **Slice Displaying the Four Corners of an Array**

## Example 3

You can use a stride to invert the order *and* skip elements. For example, here is a slice that begins with the upper bound of the array and display every other element until it reaches the lower bound of the array: **(::–2)**. Thus, using **(::–2)** with a Fortran **integer(10)** array tells TotalView to displays the following elements:

```
(10)
(8)
(6)
...
```

## Example 4

You can also combine inverse order and a limited extent to display a small section of a large array. The following example specifies a **(2:3,7::–1)** slice with a **integer*4(–1:5,2:10)** Fortran array:

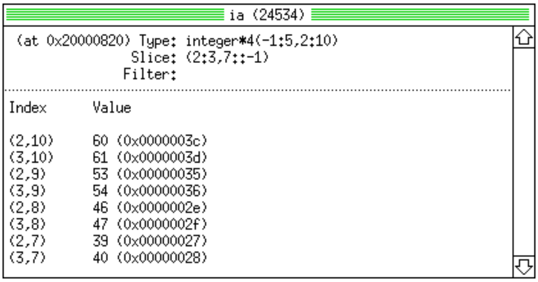



FIGURE 72: **Fortran Array with Inverse Order and Limited Extent**

As you can see, TotalView only shows elements in rows 2 and 3 of the array, beginning with column 10 and ending with column 7.

## Using Slices in the Variable Command

When you use the **Variable** command to display a Variable Window, you can include a slice expression as part of the variable name. Specifically, if you include an array name followed by a set of slice descriptions in the variable dialog box, TotalView initializes the slice field in the Variable Window to the slice descriptions that you specified.

If you include an array name followed by a list of subscripts in the variable dialog box, TotalView interprets the subscripts as a slice description rather than as a request to display an individual value of the array. As a result, you can display different values of the array by changing the slice expression.

For example, suppose that you have a 20-element by 10-element Fortran array named **array2**, and you want to display element **(5,5)**. Using the **Variable** command, you enter **array2(5,5)**. This sets the initial slice to **(5:5,5:5)**.

You can tell TotalView to display one of the array's values by enclosing the array name and subscripts (that is, the information normally included in a slice expression) within parentheses, such as **(array2(5,5))**. In this case, the Variable Window just displays the type and value of the element and does not show its array index. See Figure 73.

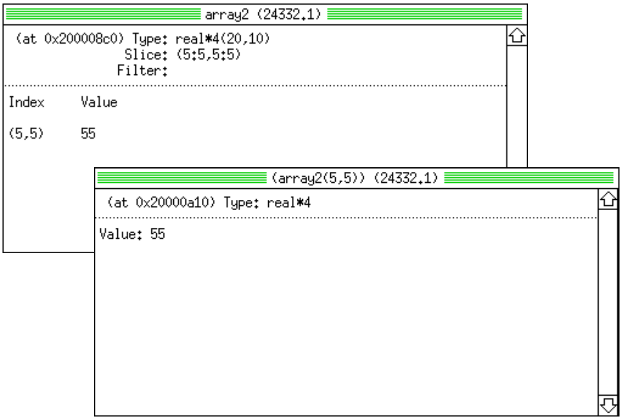



FIGURE 73: **Variable Window for *array2***

The top figure shows the information displayed for **array2(5:5)**. The bottom figure shows the information for **(array2(5:5))**.

## Array Data Filtering

You can filter arrays of type character, integer, or floating point by specifying a filter expression in the **Filter** field. Your filtering options are:

- Arithmetic comparison to a constant value
- Equal or not equal comparison to IEEE NANs, INFs, and DENORMs
- Within a range of values, inclusive or exclusive
- General expressions

When an element of an array matches the filter expression, the element is included in the Variable Window display.

You can also sort array elements into an ascending or descending order and display statistical information about the array.

## Filtering by Comparison

Specify arithmetic comparisons to a constant value with the following format:

*operator value*

where *operator* is either a C/C++ or Fortran style comparison operator, and *value* is a signed or unsigned integer constant, or a floating-point number. Table 23 lists the comparison operators.

TABLE 23: Array Data Filtering Comparison Operators

| Comparison | C/C++ Operator | Fortran Operator |
|---|---|---|
| Equal | == | .eq. |
| Not equal | != | .ne. |
| Less than | < | .lt. |
| Less than or equal | <= | .le. |
| Greater than | > | .gt. |
| Greater than or equal | >= | .ge. |

The following figure shows an array whose filter is "<100". This indicates that TotalView should only display array elements whose value is less than 100.

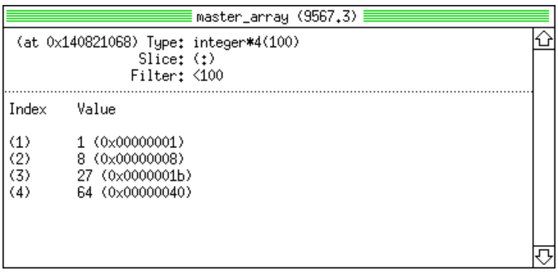



FIGURE 74: **Array Data Filtering by Comparison**

If the *value* you are using in the comparison is an integer constant, TotalView uses a signed comparison. If you add a **u** or **U** to the constant, TotalView performs an unsigned comparison.

## Filtering for IEEE Values

You can filter IEEE NaN, infinity, or denormalized floating-point values by specifying a filter in the following form:

*operator ieee-tag*

For *operator*, only the *equal* and *not equal* comparison operators listed in Table 23 are allowed.

The *ieee-tag* represents an encoding of IEEE floating point values, as explained in the following table.

TABLE 24: Array Data Filtering IEEE Tag Values

| IEEE Tag Value | Meaning |
|---|---|
| $nan | NaN (Not a number), either Quiet or Signaling |
| $nanq | Quiet NaN |
| $nans | Signaling NaN |
| $inf | Infinity, either Positive or Negative |
| $pinf | Positive Infinity |
| $ninf | Negative Infinity |
| $denorm | Denormalized number, either positive or negative |
| $pdenorm | Positive denormalized number |
| $ndenorm | Negative denormalized number |

Figure 75 shows an example of filtering an array for IEEE values. The top left figure shows how TotalView displays the unfiltered array. Notice the INF, -INF, NANQ, and NANS values. This is followed by two filtered displays. The first only shows the values of denormalized numbers. The second only shows infinite values.

## Filtering by Range of Values

Specify range expressions using the format:

[>] *low-value* : [<] *high-value*

where *low-value* specifies the lowest value to include, and *high-value* specifies the highest value to include, separated by a colon. By default, the high and

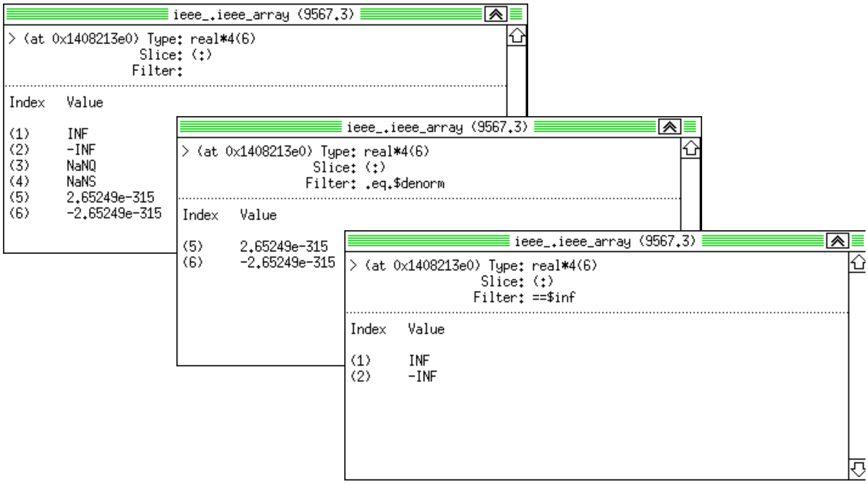



FIGURE 75: **Array Data Filtering for IEEE Values**

low values are inclusive. If you specify a > before *low-value*, the low value is exclusive. Similarly, a < before the *high-value* makes it exclusive.

The *low-value* and *high-value* must be constants of type integer, unsigned integer, or floating-point. The type of *low-value* must be the same as the type of *high-value*, and *low-value* must be less than *high-value*. If *low-value* and *high-value* are integer constants, they can be immediately followed by **u** or **U**, to force an unsigned comparison. The following figure shows a filter applied to an array such that only values equal to or greater than 64 but less than 512 are displayed.

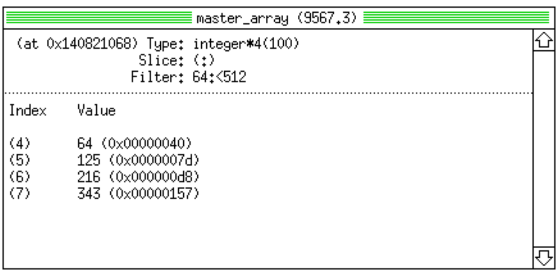



FIGURE 76: **Array Data Filter by Range of Values**

## Array Filter Expressions

The filtering capabilities described in the previous sections are those that are most often used. In some circumstances, you may want to create more general filter expressions. When you create a filter expression, you are creating a Fortran or C Boolean expression that TotalView evaluates for every element in the array or the array slice. For example, here is an expression that displays all array elements whose contents are greater than 0 and less than 50 or greater than or 100 and less than 150.

```
($value > 0 && $value < 50) ||
                ($value > 100 && $value < 150)
```

As TotalView looks at array elements, it sets the **$value** special variable to the element's value. It then evaluates your expression. So, if your array had 15 elements, this expression would be evaluated 15 times.

Notice also the use of the **&&** operator to "and" the two parts of the Boolean expression together. You can use any of TotalView's standard operators. And, the way in which TotalView computes the results of an expression is identical to the way it computes values at an evaluation point. For more information, see "*Defining Evaluation Points*" on page 213.

**NOTE** However, you cannot use any of the IEEE tag values described in "*Filtering for IEEE Values*" on page 173.

## Filter Comparisons

TotalView lets you filter array information in a variety of ways—and these ways can overlap. For example, the following two filters produce the same result:

```
> 100
$value > 100
```

Similar, you obtain the same results with either of the following:

```
>0:<100
$value > 0 && $value < 100
```

In both of these, the first method is easier to type than the second. In general, you would use the second method if you were creating more complicated expressions.

### Filtering Array Data

The procedure for filtering an array is quite simple: select the **Filter** field, enter the array filter expression, and then press Return.

The Variable Window is updated to include only the elements that match the filter expression.

TotalView applies the filter expression to each element of the array after any array slice is applied. If the value of an element matches the filter expression, TotalView displays the element.

If necessary, TotalView converts the array element before evaluating the filter expression. The following conversion rules apply:

- If the filter operand or array element type is floating-point, TotalView converts it to a double precision floating-point value. Extended precision values are truncated to double precision. Converting integer or unsigned integer values to double precision values may result in a loss of precision. Unsigned integer values are converted to non-negative double precision values.
- If the filter operand or the array element is an unsigned integer, TotalView converts the values to unsigned 64-bit integer.
- If both the filter operand and array element are of type integer, TotalView converts the values to type 64-bit integer.

These conversions modify a copy of the array's elements—they never alter the actual array elements.

To stop filtering an array, delete the contents of the **Filter** field in the Variable Window and press Return. TotalView will then update the Variable Window so that it includes all elements.

## Sorting Array Data

TotalView lets you sort the displayed array data into ascending or descending order. (It does not, of course, sort the actual data.) The sort commands appear within the popup menu that you can display from within the Variable Window. (See Figure 77.)

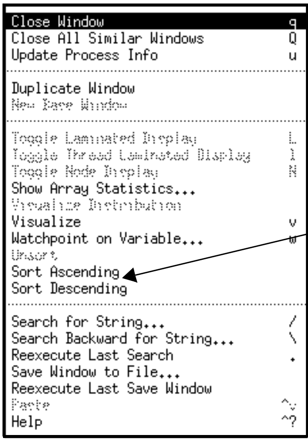



FIGURE 77: **Sort Items on the Process Pop Up Menu**

If you select **Sort Ascending,** TotalView places all of the array's elements in ascending order. For example:

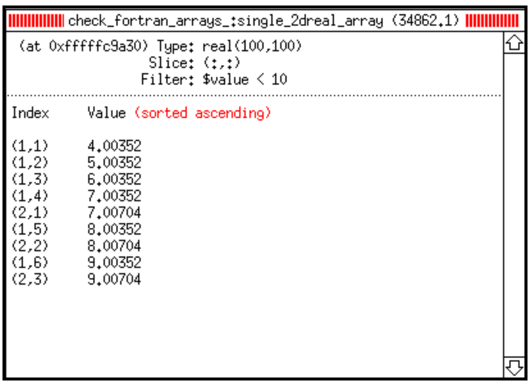



FIGURE 78: **Sort Window**

As you would expect, **Sort Descending** places array elements into a descending order. The **Unsort** command (located above the **Sort Ascending** and **Sort Descending** commands) returns the array to its original order.

The sort commands only manipulate the displayed elements. This means that if you limit the number of elements by defining a slice or a filter, the debugger only sorts the elements displayed by the slicing or filtering action.

## Array Statistics

The **Show Array Statistics** command, which is also found on the Variable popup menu, displays a window containing information about your array. Here is an example:

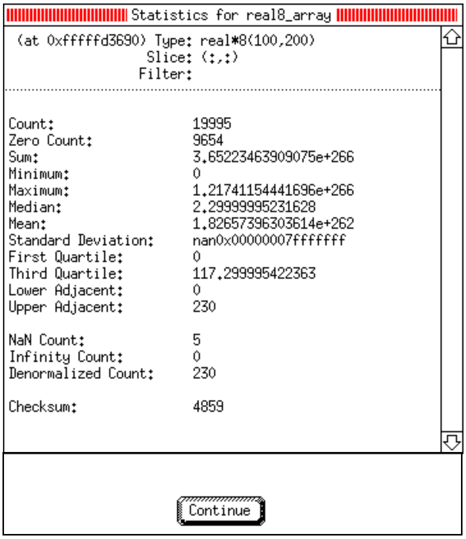



FIGURE 79: **Array Statistics Window**

If you have added a filter or a slice, these statistics only describe the information that is being displayed; the statistics do not describe the entire unfiltered array.

The statistics that are displayed are as follows:

- **Checksum**

  A checksum value for the array elements.

- **Count**

  The total number of displayed array values. If you are displaying a floating point array, this number does not include NaN or Infinity values.

- **Denormalized Count**

  A count of the number of denormalized values found in a floating point array. This includes both negative and positive denormalized values as defined in the IEEE floating point standard. Unlike other floating point statistics, these elements participate in the statistical calculations. (This entry only appears if a floating point array is being displayed.)

- **Infinity Count**

  A count of the number of infinity values found in a floating point array. This includes both negative and positive infinity as defined in the IEEE floating point standard. These elements do not participate in the statistical calculations. (This entry only appears if a floating point array is being displayed.)

- **Lower Adjacent**

  This value provides an estimate of the lower limit of the distribution. Values below this limit are called *outliers*. The lower adjacent value is the first quartile value less 1.5 times the difference between the first and third quartiles.

- **Maximum**

  The largest array value.

- **Mean**

  The average value of array elements.

- **Median**

  The middle value. Half of the array's values are less than the median and half are greater than the median.

- **Minimum**

  The smallest array value.

- **NaN Count**

  A count of the number of NaN values found in a floating point array. This includes both signaling and quiet NaNs as defined in the IEEE floating point standard. These elements do not participate in the statistical calculations. (This entry only appears if a floating point array is being displayed.)

- **Quartiles, First and Third**

  Either the 25th or 75th percentile values. The first quartile value means that 25% of the array's values are less than this value and 75% are greater than this value. In contrast, the fourth quartile value means that 75% of the array's values are less than this value and 25% are greater.

- **Standard Deviation**

  The standard deviation of the array's values.

- **Sum**

  The "sum" of all of the displayed array's values.

- **Upper Adjacent**

  This value provides an estimate of the upper limit of the distribution. Values above this limit are called *outliers*. The upper adjacent value is the third quartile value plus 1.5 times the difference between the first and third quartiles.

- **Zero Count**

  The number of elements having whose value is 0.

## Displaying a Variable in All Processes or Threads

When you are debugging a parallel program that is running many instances of the same executable, or a multithreaded program, you usually need to view or update the value of a variable in all of the processes (or threads) at once.

To display the value of a variable in all of the processes in a parallel program, first bring up a Data Pane displaying the value of a variable in one of the processes. You can now use the:

- **Toggle Laminated Display** command from the Data Pane menu to display the value of the variable in all of the processes.
- **Toggle Thread Laminated Display** command to display the value of a variable in all threads within a single process.

**NOTE** You cannot laminate across processes and threads in the same data page simultaneously.

The Data Pane switches to "laminated" mode, and displays the value of the variable in each process or thread. Figure 80 shows a simple, scalar variable in each of four processes of an MPI code. In the top window, all of the processes have the variable in a matching stack frame, so the value is displayed for all of them. In the bottom window, a corresponding variable cannot be found, so that information is displayed in a Data Pane.

The top figure shows a laminated scalar. The bottom shows a laminated scalar with missing call frames in some processes.

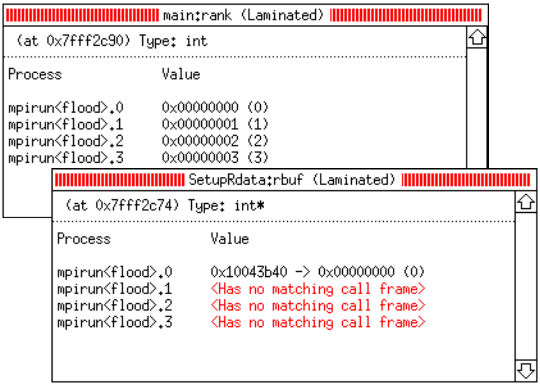



FIGURE 80: **Laminated Scalar Variable**

If you decide that you no longer want the pane to be laminated, use the same command to delaminate it, and return it to being a normal Data Pane.

When looking for a matching stack frame, TotalView matches frames starting from the top frame, and considers calls from different memory or stack locations to be different calls. For example:

```
int recurse (int i, int depth)
{
    if (i == 0)
        return depth;
    if (i&1)
        recurse (i-1, depth+1);
    else
        recurse (i-3, depth+1);
}
```

The two calls to **recurse** generate non-matching stack frames.

If the variables are at different addresses in the different processes or threads, the address field at the top of the pane displays **(Multiple)** and the unique addresses are displayed with each data item, as shown in Figure 81.

TotalView also allows you to laminate arrays and structures. When you laminate an array, each element in the array is displayed across all processors. As with a normal Data Pane, you can use a slice to select elements to be displayed. Figure 82 shows an example of a laminated array and a laminated structure. You can also laminate an array of structures.

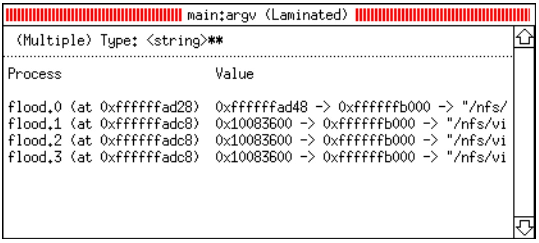



FIGURE 81: **Laminated Variable at Different Addresses**


❶ Laminated array
❷ Element [0] for each of the processes
❸ Structure elements for one process
❹ Laminated structure

FIGURE 82: **Laminated Array and Structure**

## Diving in a Laminated Pane

You can dive through pointers in a Laminated Data Pane, and the dive will apply to the associated pointer in each process or thread.

### Editing a Laminated Variable

If you edit a value in a laminated Data Pane, TotalView asks if it should apply this change to all of the processes or threads or only the one in which you made a change. This is also an easy way to update a variable in all of the processes such as a global debug flag.

## Visualizing Array Data

The TotalView Visualizer is part of a suite of software development tools for debugging, analyzing and tuning the performance of programs. It works with TotalView to create graphic images of array data in your programs. This lets you see your data in one glance and quickly find problems with it as you debug your programs.

The Visualizer is implemented as a self-contained process. It can be launched directly by TotalView to visualize data as you debug your programs. Alternatively, you can run the visualizer from the command line to visualize data dumped to a file in a previous TotalView session.

For information about running the TotalView Visualizer, see Chapter 9, "*Visualizing Data*" on page 247.

### Visualizing a Laminated Data Pane

You can export data from a laminated Data Pane to the visualizer using the Visualize command. However the process (or thread) index will be the first axis of the visualization, and therefore you must use one fewer data dimension than you normally would. If you do not want the process/thread axis to be significant in the visualization, you can use a normal Data Pane, since all of the data must necessarily be in one process.

## Displaying Thread Objects

On some platforms, TotalView can display information about thread objects, which are objects that let you coordinate your application's threads. The objects for which TotalView can display information are mutexes, condition variables, read-write locks, and pthread-specific data keys.

## Displaying Mutex Information

**NOTE** The Mutex Information Window is supported only on Compaq Tru64 UNIX and AIX systems.

A mutex is a mutual exclusion object that allows multiple threads to synchronize access to shared resources. A mutex has two states: locked and unlocked. Once a mutex is locked by a thread, other threads attempting to lock it will block. Only after a locking thread unlocks (releases) the mutex can one of the blocked threads acquire (lock) the mutex and proceed.

The Mutex Information Window contains a list of all mutexes known in a process. You can tell TotalView to display this window if you place your cursor in the Process Window and then select the **Mutex Info Window** command from the **Process State Info** menu. If you are using a Compaq Tru64 UNIX system, TotalView responds by displaying the window shown in the following figure.

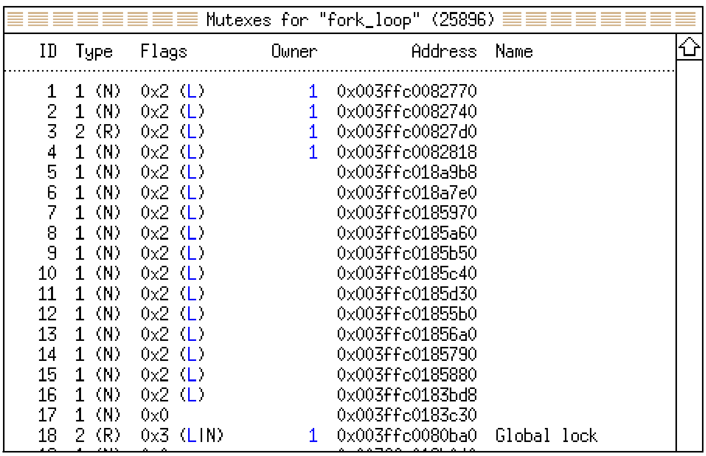
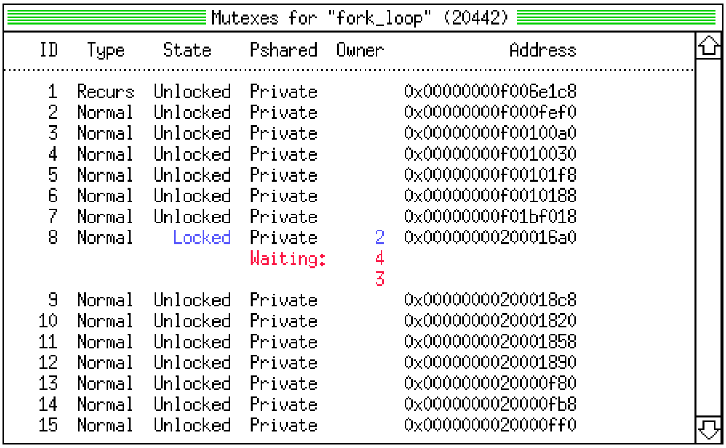



FIGURE 83: **Compaq Tru64 UNIX Mutex Info Window**

The window displayed on AIX systems is:



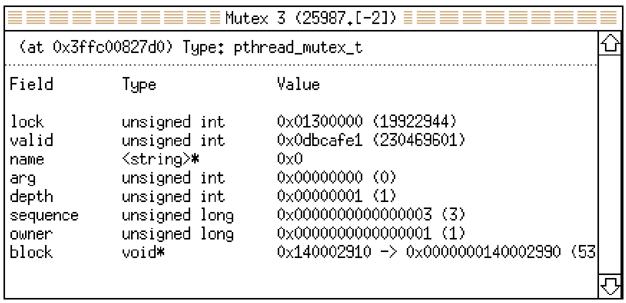
Mutexes for "fork_loop" (20442)

| ID | Type | State | Pshared | Owner | Address |
|---|---|---|---|---|---|
| 1 | Recurs | Unlocked | Private | | 0x00000000f006e1c8 |
| 2 | Normal | Unlocked | Private | | 0x00000000f000fef0 |
| 3 | Normal | Unlocked | Private | | 0x00000000f00100a0 |
| 4 | Normal | Unlocked | Private | | 0x00000000f0010030 |
| 5 | Normal | Unlocked | Private | | 0x00000000f00101f8 |
| 6 | Normal | Unlocked | Private | | 0x00000000f0010188 |
| 7 | Normal | Unlocked | Private | | 0x00000000f01bf018 |
| 8 | Normal | Locked | Private | 2 | 0x00000000200016a0 |
| | | | Waiting: | 4 | |
| | | | | 3 | |
| 9 | Normal | Unlocked | Private | | 0x00000000200018c8 |
| 10 | Normal | Unlocked | Private | | 0x0000000020001820 |
| 11 | Normal | Unlocked | Private | | 0x0000000020001858 |
| 12 | Normal | Unlocked | Private | | 0x0000000020001890 |
| 13 | Normal | Unlocked | Private | | 0x0000000020000f80 |
| 14 | Normal | Unlocked | Private | | 0x0000000020000fb8 |
| 15 | Normal | Unlocked | Private | | 0x0000000020000ff0 |

FIGURE 84: **AIX Mutex Info Window**

For each mutex, TotalView displays the following information:

**ID** The sequence number assigned to a mutex by the threads package. Diving into this field opens a data window containing a view of the mutex's data.



Mutex 3 (25987.[-2])

(at 0x3ffc00827d0) Type: pthread_mutex_t

| Field | Type | Value |
|---|---|---|
| lock | unsigned int | 0x01300000 (19922944) |
| valid | unsigned int | 0x0dbcafe1 (230469601) |
| name | <string>* | 0x0 |
| arg | unsigned int | 0x00000000 (0) |
| depth | unsigned int | 0x00000001 (1) |
| sequence | unsigned long | 0x0000000000000003 (3) |
| owner | unsigned long | 0x0000000000000001 (1) |
| block | void* | 0x140002910 -> 0x0000000140002990 (53 |

FIGURE 85: **Mutex Data Window on Compaq Tru64 UNIX**

**Type** The mutex type. These types are set using the **pthread_mutexattr_settype()** call on the attribute object before the mutex is initialized.

**Compaq Tru64 UNIX**: This is a mutex type number and a single-character abbreviation of the type name.

**N**—A normal mutex.

**R**—A recursive mutex.

**E**—An error-check mutex. Error-check mutexes contain additional information for use in debugging, such as the thread ID of the locking thread. During program development, you should use error-check mutexes in place of normal mutexes, and only switch to the simpler version when performance becomes an issue.

While your system may have other types available, TotalView only shows these three types.

**AIX**: The type is one of the following:

**Normal**—A Normal mutex.

**Recurs**—A recursive mutex.

**ErrChk**—An error check mutex.

**NRecNP**—A non-portable, non-recursive mutex.

**RcurNP**—A non-portable, recursive mutex.

**FastNP**—A non-portable fast mutex.

**Flags** (Compaq Tru64 UNIX only)

This column contains hex strings that describe the current mutex flags and a one-character abbreviation for some flags:

**0x8 (M): Metered**. The mutex contains metering information.

**0x4 (W): Waiters**. One or more threads are waiting for this mutex. By default, waiting threads are shown in red; their color is the same as the thread's *error* state flag color.

**0x2 (P): Locked**. The mutex is locked. By default, locked mutexes are shown in blue; their color is the same as the thread's *stopped* state flag color.

**0x1 (N): Name**. This mutex has a name.

While your system may use additional flag bits, TotalView only shows names for these flags.

**State** (AIX only) The mutex lock state is displayed as follows:

**Unlocked**—The mutex is unlocked.

**Locked**—The mutex is locked. By default, this is shown in blue; its color is the same as the thread's *stopped* state flag color.

**Pshared** (AIX only) This value indicates if the mutex is shared by other processes.

**Private**—The mutex can only be manipulated by threads in the process that initialized the mutex.

**Shared**—The mutex can be manipulated by any thread that has access to the mutex's memory. (Some versions of IBM's system libraries cannot provide information on shared mutexes to TotalView. If this information is not available, TotalView only describes private mutexes.)

**Owner** If the mutex is locked, this field displays the locking thread's system TID.

NOTE **On Compaq Tru64 UNIX, the owner TID is only available for error-check mutexes.**

Diving or selecting on this number tells TotalView to display the locking thread's Process Window. This is the same window that TotalView would displays if you dive or select the thread's entry in the Root Window.

If threads are waiting for this mutex, their system TIDs are shown in the owner field, with one thread ID displayed on each line. You can open a Process Window for these waiting threads by diving or selecting on its number.

NOTE **If TotalView cannot obtain this information, it does not show blocked thread lines.**

**Address** This field contains the memory address of the mutex. You can open a data window containing a view of the mutex's data by diving on this field. (This window appeared previously in this section.)

**Name** (Compaq Tru64 UNIX only)
If the mutex has a name, it is shown here. If you are using version 4.0D or later of the operating system, the **pthread_mutex_setname_np()** routine provides the mutex's name. However, this routine is not portable.

## Displaying Condition Variable Information

The window that displays the condition variables lists all the condition variables known in this process.

**NOTE** The Condition Variables Window is supported only on Compaq Tru64 UNIX and AIX systems.

You can tell TotalView to display its Condition Variables Window if you place your cursor in the Process Window and then select the **Condition Variable Info Window** command from the **Process State Info** menu. If you have an Compaq Tru64 UNIX system, TotalView displays the following window.

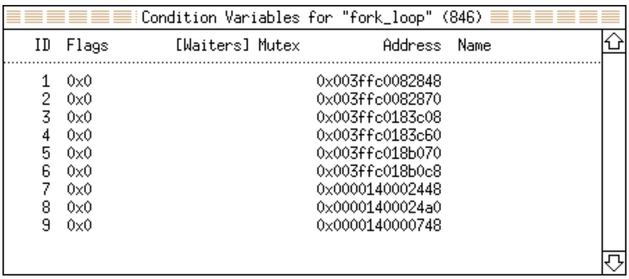



FIGURE 86: **Compaq Tru64 Condition Variable Window**

If you have an AIX system, here is the window that you will see:

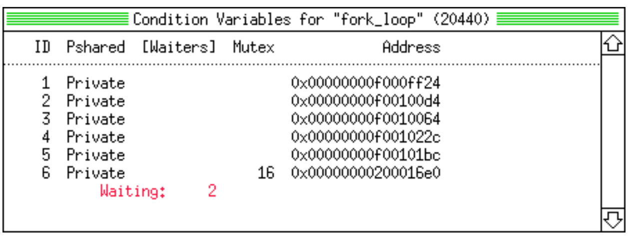
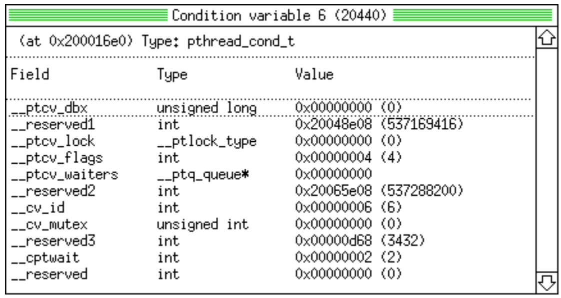



FIGURE 87: **AIX Condition Variable Window**

For each condition variable, TotalView displays the following information:

**ID** The ID is the sequence number assigned to this condition variable by the threads package. Diving into this field opens a data window containing a view of the condition variable's data. This window is shown in Figure 88.


Condition variable 6 (20440)

(at 0x200016e0) Type: pthread_cond_t

| Field | Type | Value |
|---|---|---|
| __ptcv_dbx | unsigned long | 0x00000000 (0) |
| __reserved1 | int | 0x20048e08 (537169416) |
| __ptcv_lock | __ptlock_type | 0x00000000 (0) |
| __ptcv_flags | int | 0x00000004 (4) |
| __ptcv_waiters | __ptq_queue* | 0x00000000 |
| __reserved2 | int | 0x20065e08 (537288200) |
| __cv_id | int | 0x00000006 (6) |
| __cv_mutex | unsigned int | 0x00000000 (0) |
| __reserved3 | int | 0x00000d68 (3432) |
| __cptwait | int | 0x00000002 (2) |
| __reserved | int | 0x00000000 (0) |

FIGURE 88: **Compaq Tru64 UNIX Condition Variable Data Window**

**Flags** (Compaq Tru64 UNIX only)
The information in this column is a hex string containing the current condition variable's flags and a one-character abbreviation for some of the flags:

**0x8 (M): Metered**. This condition variable contains metering information.

**0x4 (W): Waiters**. One or more threads is waiting for this condition variable. By default, this is shown in red, which is the same as the thread's *error* state flag color.

**0x2 (P): Pending**. A wakeup is pending for this condition variable. By default, this is shown in blue; its color is the same as the thread's *stopped* state flag color.

**0x1 (N): Name**. The condition variable has a name.

While your system may use more flags, TotalView only shows these four flag names.

**Pshared** (AIX only) This value indicates if the condition is shared by other processes.

**Private**—The condition value can only be manipulated by threads in the process that initialized it.

**Shared**—The condition value can be manipulated by any thread that has access to its memory. (Some versions of IBM's system libraries cannot provide information on shared condition values to TotalView. If this information is not available, TotalView only describes private condition values.)

**Waiters** If threads are waiting for this condition variable, the debugger displays their system TIDs, one thread for each line, on the lines following the condition variable. Diving or selecting entries in the list of waiting threads open windows for them.

> NOTE **If TotalView cannot obtain this information, it does not show waiting threads.**

**Mutex** This field contains the ID of the mutex that guards the condition variable. If TotalView can translate the ID into an address, diving into this field opens a data window containing a view of the guard mutex's data.

TotalView can only translate this ID if it was correctly initialized. That can be done statically or by using an attributes object. See the mutex and condition variable **man** pages for more information.

**Address** This field has the condition variable's memory address. Diving into the address field opens a data window containing a view of the actual condition variable's data.

**Name** (Compaq Tru64 UNIX only)
If the condition variable has a name, it is shown here. If you are using version 4.0D or later of the operating system, the **pthread_mutex_setname_np()** routine provides the condition variable's name. However, this routine is not portable.

## Displaying Read-Write Lock Information

**NOTE** The Read-Write Lock Information Window is supported only on AIX systems.

A read-write lock is a mutual exclusion object that allows multiple threads to synchronize access to shared resources. A read-write lock has three states: free, read-locked, and write-locked. A free lock can be locked by any number of readers or by one writer. Once a read-write lock is locked by a thread for one kind of access, other threads attempting to lock it for the other kind of access will block. When locking threads unlock (release) the read-write lock, blocked threads can acquire (lock) it and proceed.

The Read-write Lock Information Window contains a list of all read-write locks known in this process. You can tell TotalView to display its Read-write Lock Window if you place your cursor in the Process Window and then select the **Read-Write Lock Info Window** command from the **Process State Info** menu. TotalView responds by showing in the following figure.

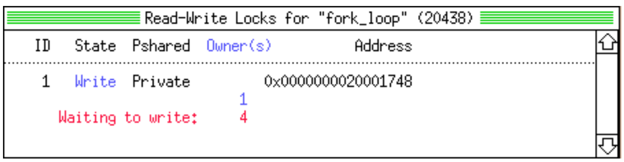



FIGURE 89: **Read-Write Lock Info Window**

For each read-write lock, TotalView displays the following information:

**ID** This field contains the sequence number assigned to this read-write lock by the threads package. Diving into this field opens a data window containing a view of the actual read-write lock data.

**State** This field displays the read-write lock state as follows:

**Free**—Unlocked.

**Read**—Locked for reading. By default, this is shown in blue; its color is the same as the thread's *stopped* state flag color.

**Write**—Locked for writing. By default, this is shown in blue; its color is the same as the thread's *stopped* state flag color.

**Pshared**

This value indicates if the read-write lock is shared by other processes.

**Private**—The read-write lock can only be manipulated by threads in the process that initialized it.

**Shared**—The read-write lock can be manipulated by any thread that has access to its memory. (Some versions of IBM's system libraries cannot provide information on shared read-write locks to TotalView. If this information is not available, TotalView only describes private read-write locks.)

**Owner**

If the read-write lock is locked, this field displays the system TID of the locking thread. Diving or selecting on this number tells TotalView to display the Process Window for the locking thread. TotalView displays the same window if you dive or select the thread's entry in the Root Window.

If threads are waiting for this read-write lock, their system TIDs are shown in the owner field, with one thread ID being displayed for each line in the window. You can open a Process Window for a waiting thread by diving or selecting its number.

If TotalView cannot obtain this information, it does not show blocked thread lines.

NOTE **Some versions of IBM's system libraries cannot provide the correct owner TID for read-write locks locked for reading. In these cases, the owner TID can only be trusted when it the lock is in its write state.**

**Address**

The memory address of the read-write lock. You can open a data window containing a view of the read-write lock data by diving on this field. The window TotalView displays is as follows:

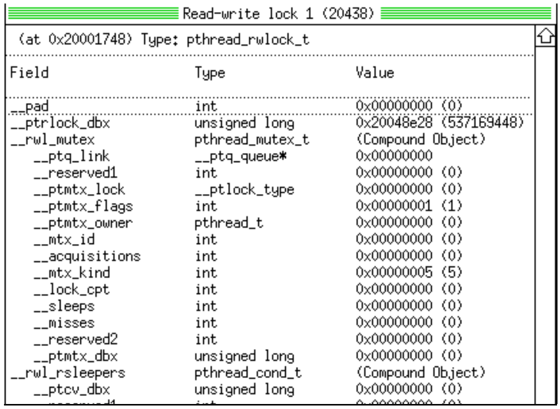



FIGURE 90: **Read-Write Lock Data Window**

## Displaying PThread-Specific Data Key Information

**NOTE** The Key List Window is supported only on AIX systems.

A pthread-specific data key is an object that can have a distinct pointer value of type **void** * associated with it for each pthread in a process.

The Key List Information Window contains a list of all keys known in this process. You can tell TotalView to display its Key List Window if you place your cursor in the Process Window and then select the **Key Info Window** command from the **Process State Info** menu. TotalView responds by displaying the following figure.

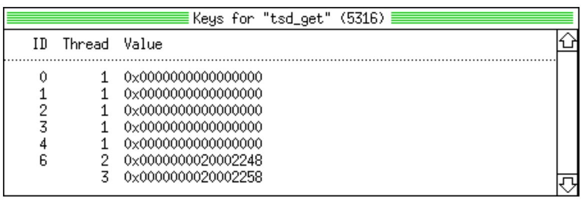



FIGURE 91: **Key List Window**

TotalView displays information for each key. Many applications initially set keys to zero (which is the NULL pointer value). Note that a key's information is not displayed until a thread sets a value for it, even if the value set is NULL.

**ID** This field contains the sequence number assigned to this key by the threads package. Only the line for the first thread's value for a key will contain an ID; subsequent lines for the same key omit the ID as a way of visually grouping values with the same ID.

**Thread** This field has the system TID of the thread that has a value for this key. Diving or selecting on this number tells TotalView to display the Process Window for the thread. TotalView displays the same window if you dive or select the thread's entry in the Root Window.

**Value** This field contains the contents of the key for a pthread. Diving into this field opens a data window containing a view of the actual key data.

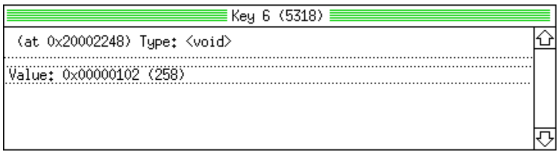



FIGURE 92: **Key Data Window**

Chapter 8

# Setting Action Points

This chapter explains how to use action points. TotalView supports four kinds of action points: breakpoints, process barrier breakpoints, evaluation points, and watchpoint. A *breakpoint* stops execution of processes and threads that reach it. A *process barrier* breakpoint holds each process that reaches it until all processes from the group reach it. An *evaluation point* causes a code fragment to execute when it is reached. A *watchpoint* lets you monitor a location in memory and stop execution when the value stored in memory is modified.

In this chapter, you'll learn how to:

- Set breakpoints
- Set evaluation points
- Set conditional breakpoints
- Set watchpoints
- Patch programs
- Set process barrier breakpoints
- Choose between interpreting and compiling expressions
- Control action points
- Save action points in a file
- Evaluate expressions
- Write code fragments
- Write assembler code (Compaq Tru64 UNIX, IBM AIX, and SGI IRIX systems only)

# Action Points Overview

Actions points allow you to specify an action that will be performed when a thread or process reaches a source line or machine instruction in your program. TotalView supports the following types of action points:

- **Breakpoints**

  When a thread or process encounters a breakpoint, it stops at the breakpoint along with the other threads in the process. You can also arrange for other related processes to stop when the breakpoint is hit.

  Breakpoints are the simplest type of action point.
- **Process barrier breakpoints**

  Process barrier breakpoints are similar to simple breakpoints, differing in that they are used to synchronize a group of processes in a multiprocess program. Process barrier breakpoints work together with the TotalView hold and release process feature.
- **Evaluation points**

  An evaluation point is a breakpoint that has a code fragment associated with it. When a thread or process encounters an evaluation point, it executes this code. Evaluation points can be used in several different ways, including conditional breakpoints, thread-specific breakpoints, countdown breakpoints, and patching code fragments into and out of your program.
- **Watchpoints**

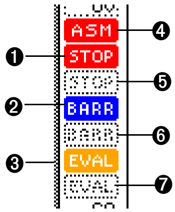
  A watchpoint lets you indicate that if a location in memory changes, TotalView should perform one of the following kinds of actions: stop the thread so that you can interact with your program or evaluate an expression. The first kind of watchpoint is analogous to a breakpoint; the second is analogous to an evaluation point.

All action points share some common properties. They:

- Can be enabled or disabled independently. A disabled action still exists; however, when your program reaches a disabled point, it continues executing.
- Can be shared across multiple processes, or set in individual processes.
- Apply to the process, so in a multithreaded process, it applies to all of the threads.

- Are assigned unique action point ID numbers. They appear in several places, including: the Root Window, the Action Points Pane of the Process Window, and the **Action Points Options** dialog box.

Each type of action point has a unique symbol. The following figure shows examples of some enabled and disabled action points:




❶ Breakpoint
❷ Process barrier breakpoint
❸ Evaluation point
❹ Assembler-level action point
❺ Disabled breakpoint
❻ Disabled barrier breakpoint
❼ Disabled evaluation point

Figure 93: Action Point Symbol

The **ASM** icon indicates that there are one or more assembler-level action points associated with the source line.

# Setting Breakpoints and Barriers

TotalView has several options for setting breakpoints. You can set:

- Source-level breakpoints
- Machine-level breakpoints
- Breakpoints that are shared among all processes in multiprocess programs

You can also control whether or not TotalView stops all processes in the program group when a single member reaches a breakpoint.

**NOTE** Breakpoints apply to the entire process, not just to a single thread. If any thread executing in the process reaches the breakpoint, TotalView will stop the process.

## Setting Source-Level Breakpoints

Typically, you set and clear breakpoints before you start a process. To set a source-level breakpoint, select a boxed line number in the tag field of the Process Window. (A boxed line number indicates that the line is associated with executable code. A **STOP** icon, as is shown in the following figure, lets you know that a breakpoint is set immediately before the source statement.


❶ Breakpoint
❷ Boxed number
❸ Gridget

Figure 94: Breakpoint Symbol

You can also set a breakpoint while a process is running by selecting a boxed line number in the tag field of the Process Window. If you set a breakpoint while the process is running, TotalView temporarily stops the process so it can insert the breakpoint and then continues running.

## Selecting Ambiguous Source Lines

If you are using C++ templates, a single source line could generate multiple function instances. If you attempt to set a source-level breakpoint by selecting a line number in a function template, and that line number has more than one instantiation, TotalView will prompt you with an **Ambiguous Source Line Selection** dialog box, as shown in Figure 95.

Use the following procedure to resolve the ambiguity.

1 Select functions by checking **All**, which selects all functions, **None**, which deselects all functions, or individual checkboxes, which lets you select and deselect functions.

  TotalView places the function name within the **Function specification** area when one box is checked. Selecting additional function prototypes clears this field.

2 Select one of the following:

  **Toggle**, which changes the state of the action points.

  **Enable**, which enables the action points, or create breakpoints or process barrier breakpoints for any that did not already exist.

  **Disable**, which disables the action point.

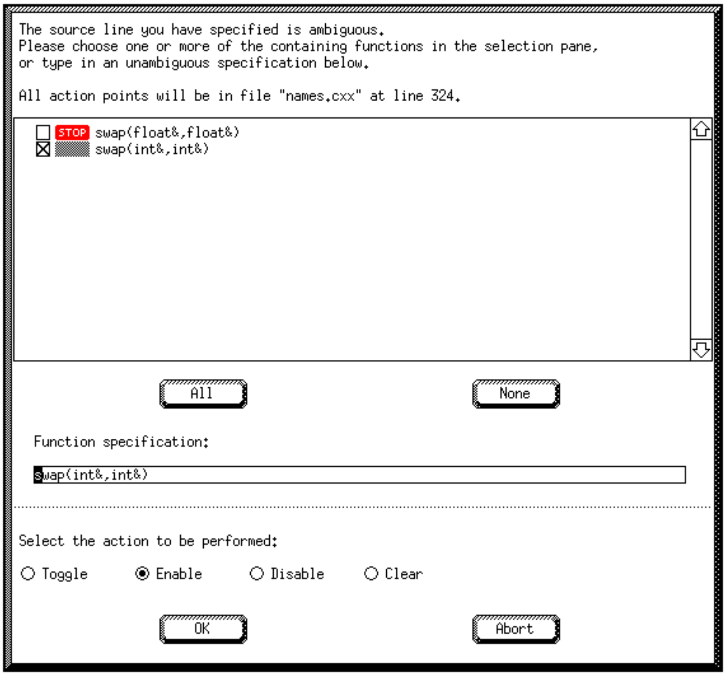



Figure 95: Ambiguous Source Line Selection Dialog Box

**Clear**, which deletes default breakpoints or process barrier breakpoints, and disable others.

3 Select the **OK** button or press Return to perform the action. If you hold down the Shift key, TotalView performs the action for process barrier breakpoints.

## Diving into Ambiguous Source Lines

Similar to selecting an ambiguous source line, if you dive on an ambiguous source line, TotalView displays the **Ambiguous Source Line** dialog box, shown in Figure 96, before it displays the **Action Point Options** dialog box.

The procedure for resolving ambiguous source lines is similar to the procedure described in "*Selecting Ambiguous Source Lines*" on page 198.

As with other action point function menus, you can specify more than one function. However, if you do, the Referenced current source lines either must not contain action points, or must contain action points of the same

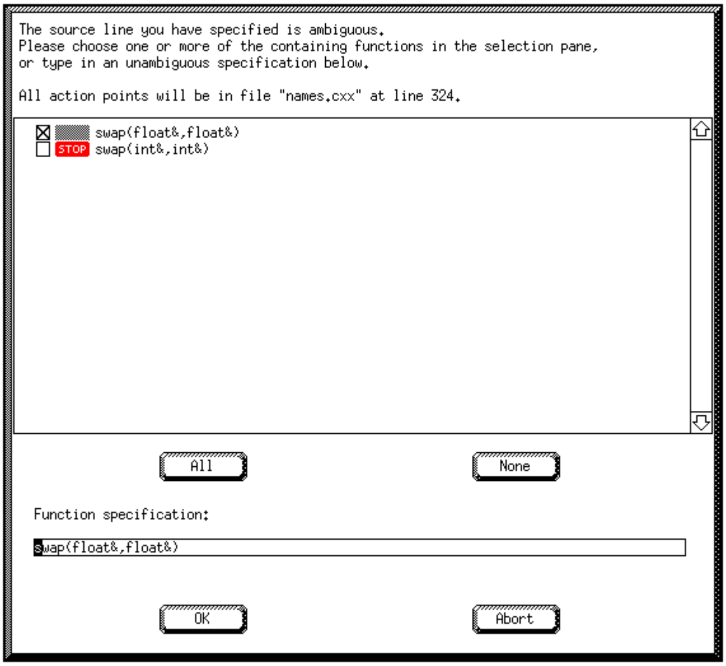



Figure 96: Ambiguous Source Line Dive Dialog Box

type. This is because the **Action Point Options** dialog box appears, and the selections you make in it apply to all selected action points.

## Toggling Breakpoints at Locations

You can toggle a breakpoint at a specific function or source line number without having to first find the function or source line in the Source Code Pane using the following procedure:

**1** Invoke the **Breakpoint at Location** command in the **STOP/BARR/EVAL/ELOG** menu of the Process Window. The **Toggle breakpoint** dialog box appears (as shown in Figure 97).

**2** Enter the name of the function or a source line number.

Entering a function name tells TotalView to toggle the breakpoint at the function's first executable source line. Entering a source line number toggles the breakpoint at the source line in the current source file.

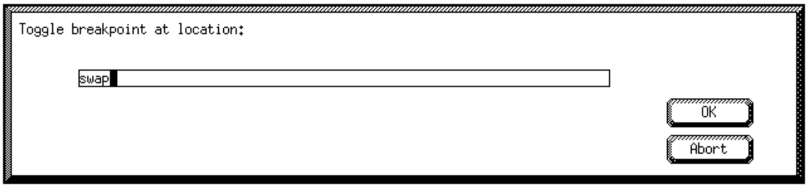



Figure 97: Toggle Breakpoint at Location Dialog Box

3 Select **OK** or press Return. If you hold down the Shift key, this command toggles a process barrier breakpoint at this location.

The behavior of the **Breakpoint at Location** command depends on whether an action point already exists at the selected location, and whether you hold down the Shift key when you select **OK** or press Return, as described in the following table.

Table 25: Breakpoint at Location Actions

| Location Content | OK Action | Shift-OK Action |
|---|---|---|
| Empty | Create **STOP** | Create **BARR** |
| **STOP** | Delete/disable **STOP** | Convert to **BARR** |
| **BARR** | Delete/disable **BARR** | Convert to **STOP** |
| **EVAL** | Disable **EVAL** | Disable **EVAL** |

### Ambiguous Locations

If you enter an ambiguous function name with the **Breakpoint at Location** command, TotalView displays its **Ambiguous Function Name** dialog box (see Figure 98 on page 202).

The procedure for resolving ambiguous function names is similar to the procedure described in "*Selecting Ambiguous Source Lines*" on page 198.

## Setting Machine-Level Breakpoints

To set a machine-level breakpoint, you must first display assembler code or source interleaved with assembler. (Refer to "*Examining Source and Assembler Code*" on page 118 for information.) You can now select the tag field opposite an instruction. The tag field must contain a gridget—the gridget indi-

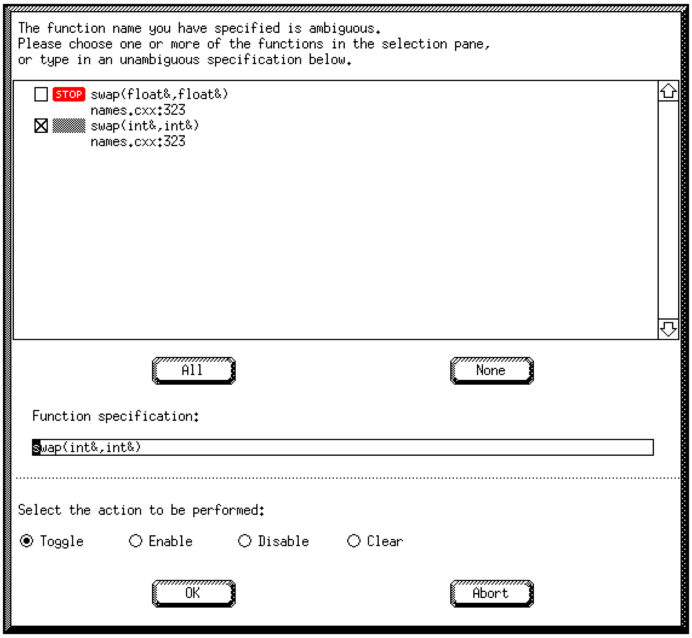



Figure 98: Ambiguous Function Name Dialog Box

cates the line is the beginning of a machine instruction. Since instruction sets on some platforms support variable-length instructions, you may see multiple lines associated with a single gridget. The **STOP** icon appears, indicating that the breakpoint occurs before the instruction is executed.

When the Source Code Pane displays source interleaved with assembler, source statements are treated as if they were comments: they are not treated as executable statements. Because they are treated as comments, you cannot set breakpoints on them. If you set a breakpoint on the first instruction after a source statement, however, you are actually creating a source-level breakpoint.

If you set machine-level breakpoints on one or more instructions that are part of a single source line and then display source code in the Source Code Pane, TotalView displays an **ASM** icon (see Figure 93) on the line number. To see the specific breakpoints, you must display assembler or assembler interleaved with source code.

When a process reaches a breakpoint, TotalView:

- Suspends the process.
- Displays the PC symbol over the stop sign to indicate that the PC currently points to the breakpoint.
- Displays **at breakpoint** in the Process Window title bar and other windows.
- Updates the Stack Trace and Frame Panes and Variable Windows.

## Thread-Specific Breakpoints

TotalView implements thread-specific breakpoints through evaluation points in the TotalView expression system. The expression system has several intrinsic variables that allow a thread to retrieve its thread ID. For example, the following example sets a breakpoint that stops the process only when thread 3 executes the evaluation point:

```
/* Stop when thread 3 evaluates this expression. */
if ($tid == 3) $stop;
```

## Breakpoints for Multiple Processes

In multiprocess programs, you can set breakpoints in the parent process and child processes before you start the program and at any time during its execution. To do this, use the **Action Point Options** dialog box, as shown in Figure 99. This dialog box provides three checkboxes for process groups:

- **Stop All Related Processes when Breakpoint Reached**

  If selected, stops all members of the program group when the breakpoint is reached. Otherwise, only the process that reaches the breakpoint stops.
- **Stop All Related Processes when Barrier Breakpoint Hit**

  If selected, stops all members of the program group when the barrier breakpoint is reached. Otherwise, only the process reaching the barrier stops.
- **Share Action Point in All Related Processes**

  If selected, enables and disables the breakpoint in all members of the share group at the same time. If this is not selected, you must enable and disable breakpoints in each share group member individually.

You can control the default setting of these checkboxes using X resources or command line options. See Figure 99.

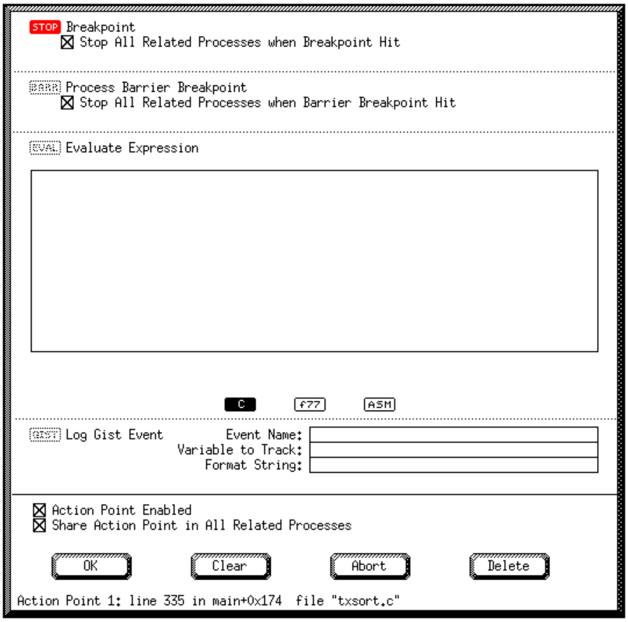



Figure 99: Action Point Options Dialog Box

The two checked boxes at the top of this figure indicate that TotalView will stop members of the program group when the action point is encountered. The check box at the bottom indicates that the action point is set in all members of the share group.

The action point ID is displayed at the bottom of the window.

For more information, refer to:

- "TOTALVIEW*STOPALL" on page 292
- "TOTALVIEW*STOPALLRELATEDPROCESSESWHENBREAKPOINTHIT" on page 292
- "TOTALVIEW*SHAREACTIONPOINT" on page 290
- "*TotalView Command Syntax*" on page 299

In addition to the controls in the **Action Point Options** dialog, you can place an expression in the expression box to control the behavior of program group members and share group members. Refer to "*Writing Code Fragments*" on page 235 for more information.

## Breakpoint when using fork()/execve()

You must link with the **dbfork** library to debug programs that call **fork()** and **execve()**. See "*Compiling Programs*" on page 11.

### Processes That Call fork()

By default, breakpoints are shared by all processes in the share group. When any process reaches a breakpoint, TotalView stops all processes in the program group. To override these defaults:

1. Dive into the tag field to display the **Action Point Options** dialog box.
2. Deselect the **Stop All Related Processes when Breakpoint Hit** and **Share Action Point in All Related Processes** checkboxes then select **OK**.

### Processes That Call execve()

Breakpoints that are shared by a parent and children with the same executable do not apply to children with different executables. To set the breakpoints for children that call **execve()**:

1. Set the breakpoints and breakpoint options desired in the parent and the children that do not call **execve()**.
2. Start the multiprocess program by displaying the **Go/Halt/Step/Next/Hold** menu and selecting the **Go Group** command. When the first child calls **execve()**, TotalView displays the following message:

   ```
   Process name has called exec (name),
   Do you wish to stop it before it enters MAIN?
   ```
3. Answer **Yes**. TotalView opens a Process Window for the process. (If you answer **No**, TotalView will not allow you to set breakpoints.)
4. Set breakpoints for the process. After you set breakpoints for the first child using this executable, TotalView does not prompt when other children call **execve()** to use it. Therefore, if you do not want to share the breakpoints among other children using the same executable, dive into the breakpoints, and set the breakpoint options.
5. Select the **Go Group** command from the **Go/Halt/Step/Next/Hold** menu to resume execution.

## Example: Multiprocess Breakpoint

The following example program illustrates the different points at which you can set breakpoints in multiprocess programs:

```
1  pid = fork();
2  if (pid == -1)
3       error ("fork failed");
4  else if (pid == 0)
5       children_play();
6  else
7       parents_work();
```

Table 26 shows the results of setting a breakpoint at different places.

Table 26: **Setting Breakpoints in Multiprocess Programs**

| Line Number | Result |
|---|---|
| 1 | Stops the parent process before it forks. |
| 2 | Stops both the parent and child processes (if the child process was successfully created). |
| 3 | Stops the parent process if **fork()** failed. |
| 5 | Stops the child process. |
| 7 | Stops the parent process. |

## Process Barrier Breakpoints

A process barrier breakpoint (process barrier point) is similar to simple breakpoint, differing in that it holds processes that reach the process barrier point. TotalView holds each process until all the processes in the group reach the same process barrier point. When the last process reaches the same barrier point, TotalView releases all processes in the group.

### Process Barrier Breakpoint States

Processes at a process barrier point are held or stopped, as follows:

**Held** A process that is *held* cannot resume execution until all the processes in its group are at the process barrier point, or until you manually release it. The various "Go" and "Single-step" commands from the **Go/Halt/Stop/Next/Hold** menu have no effect on held process.

**Stopped** When all processes in the group reach a process barrier point, TotalView automatically releases them. They remain stopped at the barrier point until you take action on them.

You can manually release held processes with the **Hold/Release Process** or **Release Group** command from the **Go/Halt/Step/Next/Hold** menu. When you manually release a process, the "Go" and "Single-step" commands become available again.

You can reuse the **Hold/Release Process** command to again toggle the hold state of the process. See "H*olding and* R*eleasing* P*rocesses*" on page 123 for more information.

## Setting a Process Barrier Breakpoint

You can set a process barrier breakpoint with the mouse or from the **Action Point Options** dialog box. To set a process barrier breakpoint with the mouse, move the mouse to the line number in the Process Window where you want to set the process barrier point. Then press **Shift-Select**.

To set a process barrier breakpoint from the **Action Point Options** dialog box, dive on the line where you want to set the process barrier point. In the **Action Point Options** dialog box, click on the **BARR** icon, then click on **OK**. See Figure 100.

If the **Stop All Related Processes when Barrier Breakpoint Hit** checkbox is selected, TotalView will stop related process when the barrier is encountered. The **Share Action Point in All Related Procecess** checkbox is automatically selected because process barrier breakpoints must be shared.

When you set a process barrier point, TotalView places it in all the processes contained within the share group.

If you run one of the processes in a group and it hits the process barrier point, you will see an **H** next to the process name in the Root Window and the word [**Held**] in the process title bar in the main Process Window. Process barrier points are always shared. See Figure 101.

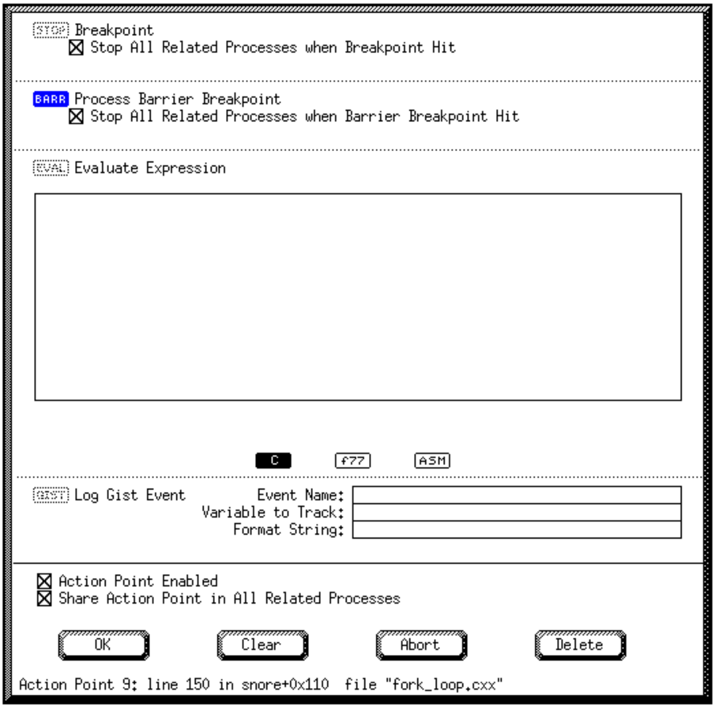



Figure 100: Action Point Options Dialog Box

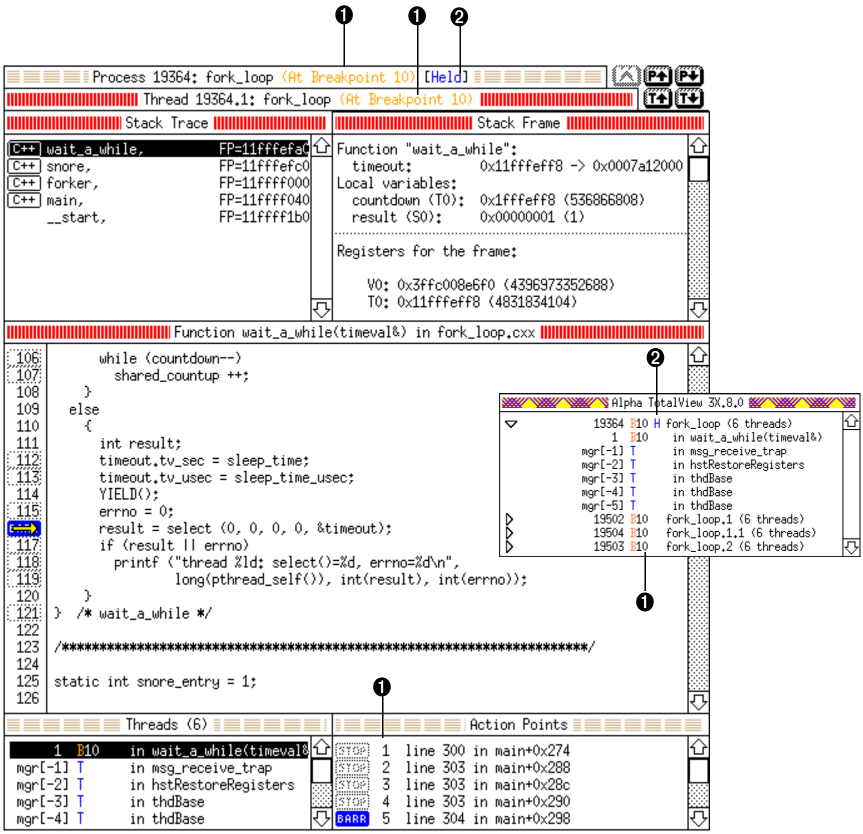
## Releasing Processes from Process Barrier Points

TotalView automatically releases processes from a process barrier point when a process hits that process barrier point and all other processes in the group are already held at it.

You can create a new process barrier point if every process in the group is already stopped at the location of the new barrier. Normally, when you create a new process barrier point TotalView holds any process stopped at the barrier's location. However, rather than holding *all* the processes in this case, TotalView does not hold any of them.

## Deleting a Process Barrier Point

You can delete a process barrier point from the **Action Point Options** dialog box or from the Process Window. If you had created the process barrier point




❶ Action point ID
❷ Hold symbol

Figure 101: **Process Barrier Breakpoint in Process and Root Windows**

using default settings, selecting the **BARR** icon in the Source Code Pane of the Process Window deletes it. In contrast, if some options were set to non-default values when you select a barrier point, TotalView just disables it. You can re-enable the barrier using its previously set options by reselecting it.

To delete a process barrier point or other action point having non-default options, dive on the action point symbol in the Source Code Pane of the Process Window to display the **Action Point Options** dialog box. In the dialog box, click **Delete**.

### Changes when Setting and Clearing a Barrier Point

Setting a process barrier point at the current PC for a *stopped* process holds the process there. If, however, all other processes in its group are at the same PC, TotalView does not hold them. Instead, TotalView treats the processes as if they were stopped at an ordinary breakpoint.

All processes that are held and which have threads at the process barrier point are released when you clear the barrier point. They remain stopped, but are no longer held. You can clear the barrier breakpoint in the **Action Point Options** dialog box by clicking on **Clear** at the bottom of the Action Points Window.

## Toggling Between a Breakpoint and a Process Barrier Point

You can convert an ordinary breakpoint to a process barrier point by moving the cursor to the breakpoint and clicking **Shift-Select**. To convert a process barrier point back to an ordinary breakpoint, move the cursor to the process barrier breakpoint and use **Shift-Select**.

Selecting a barrier point clear it in the same way that it clears breakpoints.

Note that entering **Shift-Select** on an **Eval** point does *not* convert it to a process barrier point.

## Displaying the Action Points Window

The Action Points Window displays a summary of the action points that are set in your program. To display this window, display the **STOP/BARR/EVAL/ELOG** menu and select the **Open Action Points Window** command. The Action Points Window appears, as shown in Figure 102.

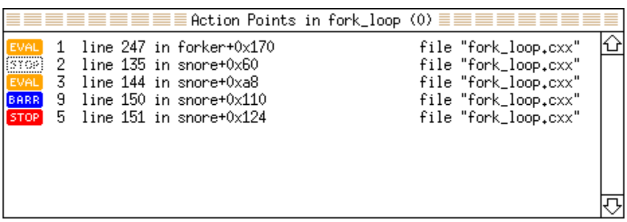



Figure 102: Action Points Window

**NOTE** The list of action points displayed in the Action Points Window is the same as shown in the Action Points Pane in the Process Window. Also, another way of getting this window is to dive on a breakpoint.

The columns in this window show:

- The type of action point
- The action point ID
- The line number
- The routine name
- The source file

Action points make it easier to navigate within your source files. You can define disabled breakpoints in your code and dive into the breakpoint to quickly display the corresponding source code in the Process Window. Thus, breakpoints can act like bookmarks in your program.

## Displaying and Controlling Action Points

The **Action Point Options** dialog box lets you set and control an action point. To display this dialog box, dive into the tag field beside a source line or an instruction. TotalView displays the dialog box shown in Figure 103 on page 212.

The following sections explain how you can control action points using the Process Window, the **Action Point Options** dialog box, and the Action Points Window.

**Disabling:** TotalView can keep an action point's definition but ignore it during execution. Disabling an action point does not remove it. TotalView remembers that an action point exists for the line, but ignores it as long as it is disabled.

You can disable an action point by:

- Deselecting **Action Point Enabled** in the **Action Point Options** dialog.
- Selecting the **STOP** or **BARR** sign in the Action Points Window.

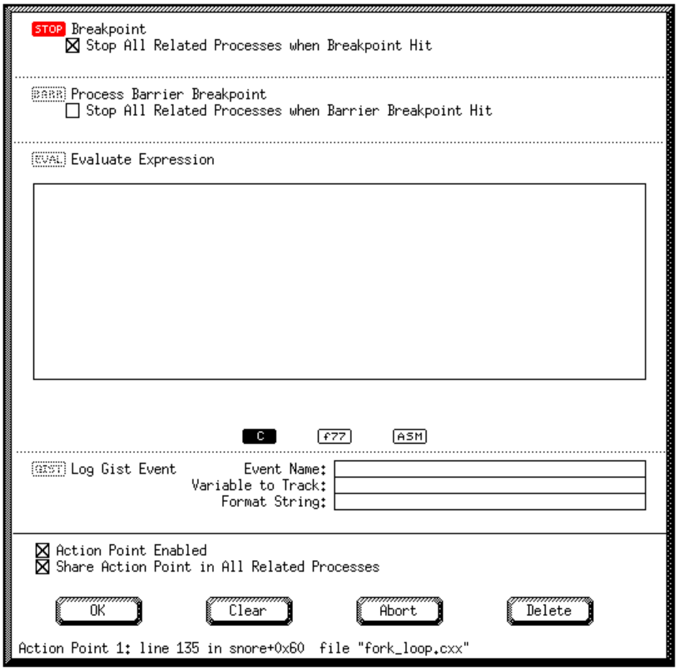



Figure 103: Action Point Options Dialog Box

**Deleting:** You can permanently remove an action point by selecting the **STOP** or **BARR** sign in the tag field or selecting the **Delete** button in the **Action Point Options** dialog.

To clear all breakpoints, and process barrier points, go to the Process Window or Action Points Window, display the **STOP/BARR/EVAL/ELOG** menu, and select the **Clear All STOP, BARR, & EVAL** command.

**Enabling:** You can activate an action point that was previously disabled by selecting the dimmed **STOP**, **BARR**, or **EVAL** sign in the process or Action Points Window, or selecting **Action Point Enabled** in the **Action Point Options** dialog.

**Suppressing:** You can tell TotalView to ignore action points during execution and prevent the creation of additional action points by selecting the **Suppress All Action Points** command on the **STOP/BARR/EVAL/ELOG**.

When you suppress action points, you disable them. In addition, you cannot update any existing action points or create new ones.

**Unsuppressing:** You can make previously suppressed action points active and allow the creation of new ones by selecting the **Unsuppress All Action Points** command on the **STOP/BARR/ EVAL/ELOG** menu.

# Defining Evaluation Points

**NOTE Assembler support is currently available on the Compaq Tru64 UNIX, IBM AIX, and SGI IRIX operating systems. Compiled expressions must be enabled to use assembler constructs.**

TotalView lets you define *evaluation points*, which are points in your program where it evaluates a code fragment. A fragment can include special commands to stop a process and its relatives. Evaluation points are often used to set *conditional breakpoints*. You can also use evaluation points to test potential fixes for your program.

You can define an evaluation point at any source line that generates executable code (marked with boxed line number in the tag field). If you display assembler or source interleaved with assembler in the Process Window, you can also define evaluation points on machine-level instructions.

As part of defining an evaluation point, you provide the code fragment to be evaluated. You can write the code fragment in C, Fortran, or assembler.

At each evaluation point, the code fragment in the evaluation point is executed before the code on that line. Typically, the program then executes the program instruction at which the evaluation point is set. But your code fragment can modify this behavior:

- It can include a branching instruction (such as GOTO in C or Fortran). The instruction can transfer control to a different point in the target program, enabling you to test program patches.
- It can contain a built-in statement. These special TotalView statements, which are called *instrinsics*, define breakpoints, process barrier points, and countdown breakpoints within the code fragment. By including them

within other statements that you code, you can define conditional breakpoints. For more information on these statements, refer to Table 31 "*Built-In Statements Used in Expressions*" on page 237.

TotalView evaluates code fragments in the context of the target program. This means that you can refer to program variables and pass control to points in the target program.

For complete information on what you can include in code fragments, refer to "*Writing Code Fragments*" on page 235.

Evaluation points only modify the processes being debugged—they do not modify the source program or create a permanent patch in the executable. If you save a program's evaluation points, however, TotalView reapplies them whenever you start a debugging session for that program. To save your evaluation points, refer to "*Saving Action Points in a File*" on page 232.

**NOTE** You should stop a process before setting an evaluation point. This ensures that the evaluation point is set in a stable context in the program.

## Setting Evaluation Points

To set an evaluation point:

1 Dive into the tag field for an instruction in the Process Window. The debugger displays the **Action Point Options** dialog box.
2 Select the **EVAL (Evaluate Expression)** button.
3 Select the button (if it's not already selected) for the language in which you will code the fragment.
4 Select the evaluation text box and enter the code fragment to be evaluated. Use the field editor commands as required. For information on supported C, Fortran, and assembler language constructs, refer to "*Writing Code Fragments*" on page 235.
5 For multiprocess programs, decide whether to share the evaluation point among all processes in the program's share group. By default, the **Share Action Point in All Related Processes** checkbox is selected for multiprocess programs, but you can override this by deselecting it.
6 Select the **OK** button to confirm your changes. If the code fragment has an error, TotalView displays an error message. Otherwise, it pro-

cesses the code, closes the dialog box, and places an **EVAL** icon in the tag field.

## Setting Conditional Breakpoints

Here are some examples of conditional breakpoints and the code fragments that you would need to supply in step 4:

- To define a breakpoint that is reached whenever a variable **i** is greater than 20 but less than 25:

```
if (i > 20 && i < 25)
    $stop;
```

- To define a breakpoint that is reached every 10th time the **$count** statement is executed:

```
$count 10
```

- To define a breakpoint with a more complex expression, consider:

```
$count i * 2
```

  When the variable **i** equals 4, the process stops the 8th time it executes the **$count** statement. After the process stops, the expression is reevaluated. If **i** now equals 5, the next stop occurs after the process executes the **$count** statement 10 more times.

For complete descriptions of the **$stop** and **$count** statements, refer to "*Built-In Statements*" on page 237.

## Patching Programs

You can use expressions in evaluation points to patch your code if you use the **goto** (C) and **GOTO** (Fortran) statements to jump to another point in your program's execution. This lets you:

- Move around code that you do not want your program to execute.
- Add new pieces of code.

In many cases, correcting an error means that you will do both operation: you patch out incorrect lines and patch in corrections.

### Conditionally Patching Out Code

For example, suppose a section of your C program dereferences a null pointer:

```
1   int check_for_error (int *error_ptr)
2   {
3        *error_ptr = global_error;
4        global_error = 0;
5        return (global_error != 0);
6   }
```

This routine calling this function assumes that the value of **error_ptr** can be 0. However, **check_for_error()** assumes that **error_ptr** is not null. Consequently, line 3 can dereference a null pointer.

You can correct this error by setting an evaluation point on line 3 and entering:

```
if (error_ptr == 0) goto 4;
```

If the value of **error_ptr** is null, line 3 is not executed.

## Patching In a Function Call

As an alternative, you can patch in a **printf()** statement that displays the value of **global_error**. You would set an evaluation point on line 4 and enter:

```
printf ("global_error is %d\n", global_error);
```

This code fragment is executed before the code on line 4; that is, it is executed before **global_error** is set to 0.

## Correcting Code

The next example contains a coding error: the function returns the maximum value instead of the minimum value:

```
1   int minimum (int a, int b)
2   {
3        int result;         /* Return the minimum */
4        if (a < b)
5             result = b;
6        else
7             result = a;
8        return (result);
9   }
```

You can correct this error by adding the following code at line 4's evaluation point:

```
if (a < b) goto 7; else goto 5;
```

This effectively replaces the **if** statement on line 4 with the statement entered at the evaluation point.

## Interpreted Versus Compiled Expressions

On most platforms, TotalView executes interpreted expressions. TotalView can also execute compiled expressions on the Compaq Tru64 UNIX, IBM AIX, and SGI IRIX platforms. On Compaq Tru64 UNIX and IBM AIX platforms, compiled expressions are enabled by default.

You can enable or disable compiled expressions using X resources or command-line options. Refer to "TOTALVIEW*COMPILEEXPRESSIONS" on page 279. See Appendix B "*Operating Systems*" on page 329 to find out how TotalView handles expressions on specific platforms.

**NOTE** Using anyone of the following intrinsics means that the evaluation point is interpreted instead of compiled: $visualize, $nid, $clid, $processduid, $duid, $tid, and $systid. In addition, $pid forces interpretation on AIX.

### Interpreted Expressions

TotalView sets a breakpoint in your code and executes the evaluation point. Since TotalView is executing the expression, interpreted expressions run slower (and possibly much slower) than compiled expressions. With multiprocess programs, interpreted expressions run more slowly because processes can be waiting for TotalView to execute the expression.

When you are debugging remote programs, interpreted expressions always run more slowly because TotalView on the host, not the TotalView debugger server (**tvdsvr**) on the client, executes the expression. For example, an interpreted expression could require that 100 remote processes wait for the TotalView debugger process on the host machine to evaluate the interpreted expression. In contrast, if the expression is compiled, it is evaluated on each remote process.

If the expression contains **$stop** or **$count**, TotalView stops evaluating the expression and stops the process. Thus, if you use **$stop** or **$count**, place them at the end of your expression because TotalView stops evaluating the expression at that point.

## Compiled expressions

TotalView compiles, links, and patches expressions into the target process. by replacing an instruction with a "branch out" instruction, relocating the original instruction, and appending the expression. This code is then executed by the target thread; this allows evaluation points and conditional breakpoints to execute very quickly. And, more importantly, this code does not need to communicate with the TotalView host process until it needs to.

If the expression contains **$stop** or **$count**, TotalView stops executing the compiled expression, so you can single step through it and continue executing the expression as you would the rest of your code. See Figure 104.

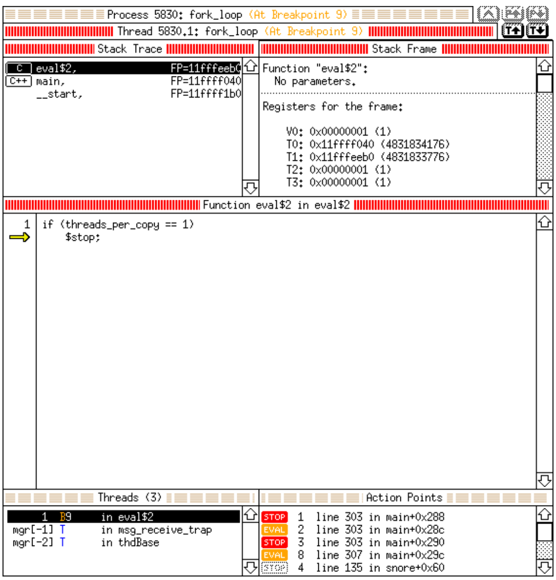



Figure 104: Stopped Execution of Compiled Expressions

If you plan to use compiled expressions, you may need to think about allocating patch space. See "*Allocating Patch Space for Compiled Expressions*" on page 220.

## Interpreted Versus Compiled Expression Performance

The greatest benefit of compiled expressions over interpreted expressions comes when either of the following two items are true.

- **The expression contains compute intensive constructs such as loops**

  Compiled machine code runs faster than interpreted expressions.

- **The number of processes and threads being debugged increases**

  For interpreted expressions, interpreting an expression in the debugger can become a bottleneck because the expression is interpreted serially on the host. In contrast, a compiled expression is executed remotely on each node without involvement by the host. And, of course, interpreted expressions execute much slower than compiled expressions and, because the interpreted expression is single-threaded through the host, your program will experience network latency delays in addition to waiting for the host to process the request.

The following table shows the performance of interpreted and compiled expressions on Compaq Tru64 UNIX, IBM AIX, and SGI IRIX platforms. A single-threaded test program used a simple loop with 15,000 iterations. An evaluation point containing a function call was placed inside the loop. The numbers shown show the mean and median intervals between consecutive occurrences of the evaluation point.

Table 27: Performance of Interpreted and Compiled Expressions

| | Interpreted (seconds) | | Compiled (seconds) | |
|---|---|---|---|---|
| Platform | Median | Mean | Median | Mean |
| Alpha | 0.057617 | 0.0787336 | 0.0 | 0.00000664 |
| AIX | 0.012652 | 0.0127334 | 0.004587 | 0.00470586 |
| IRIX | 0.011715 | 0.0209809 | 0.004347 | 0.00430252 |

All times represent the number of seconds needed to execute one evaluation point. Also, median time to execute the compiled expression on the Alpha was less than one clock tick.

## Allocating Patch Space for Compiled Expressions

TotalView must allocate or find space in your program to hold the code fragments generated by compiled expressions. Since this patch space is part of your program's address space, the location, size, and allocation scheme used by TotalView may conflict with your program. As a result, you may need to change how TotalView allocates this space. You can choose one of the following patch space allocation schemes:

- **Dynamic patch space allocation**
  Tells TotalView to find the space for the code fragment dynamically.
- **Static patch space allocation**
  Tells TotalView to use a statically allocated area of memory.

### Dynamic Patch Space Allocation

D*ynamic patch space allocation* means that TotalView allocates patch space for the code fragments dynamically. If you do not specify the size and location for the patch space, TotalView allocates 1 MB at a default location. The debugger attempts to map memory in your address space by forcing a call to the **mmap()** system call. Because this function call may not succeed, dynamic patch space allocation may not work reliably.

TotalView allocates memory for read, write, and execute access within the following addressees:

Table 28: Dynamic Patch Space Allocation Default Addresses

| Platform | Address range |
|---|---|
| Compaq Tru64 UNIX | 0xFFFFF00000 – 0xFFFFFFFFFF |
| IBM AIX | 0xCFF00000 – 0xCFFFFFFF |
| SGI IRIX (–n32) | 0x4FF00000 – 0x4FFFFFFF |
| SGI IRIX (–64) | 0x8FF00000 – 0x8FFFFFFF |

**NOTE** You can only allocate dynamic patch space for these three machines.

If the default address range conflicts with your program, or you would like to change the size of the dynamically allocated patch space, do the following:

- Specify the dynamically allocated patch space base address using the X resource "TOTALVIEW*PATCHAREAADDRESS" on page 287, or command line option "**–patch_area_base**".
- Specify the dynamically allocated patch space length using the X resource "TOTALVIEW*PATCHAREALENGTH" on page 287, or command line option "**–patch_area_length**".

## Static Patch Space Allocation

TotalView can statically allocate the patch in your program by compiling in an array with a special name. You can then specify the size of the patch space—the default size is 1 MB. TotalView looks up this special array name and uses its space as the patch space. This scheme is more reliable because TotalView will not force your program to make a function call.

To include a 1 MB statically allocated patch space in your program, add the **TVDB_patch_base_address** data object in a C module. Because this object must be 8-byte aligned, declare it as an array of doubles. For example:

```
                    /* 1 megabyte == size TV expects */
#define PATCH_LEN 0x100000
double TVDB_patch_base_address
                    [PATCH_LEN / sizeof(double)]
```

If you need to use a static patch space size that differs from the default 1 MB, you must create it in assembler language. The assembler defines two tags that TotalView uses to determine the start and end of the patch space. Since some C compilers reorder user data, you cannot reliably write this in C. Table 29 shows sample assembler code for each platform.

Table 29: **Static Patch Space Assembler Code**

| Platform | Assembler Code |
|---|---|
| Compaq Tru64 UNIX | `    .data`<br>`    .align 3`<br>`    .globl TVDB_patch_base_address`<br>`    .globl TVDB_patch_end_address`<br>`TVDB_patch_base_address:`<br>`    .byte 0x00 :` *PATCH_SIZE*<br>`TVDB_patch_end_address:` |

Table 29: Static Patch Space Assembler Code (cont.)

| Platform | Assembler Code |
|---|---|
| IBM AIX | `.csect .data{RW}, 3`<br>`.globl TVDB_patch_base_address`<br>`.globl TVDB_patch_end_address`<br>`TVDB_patch_base_address:`<br>`.space PATCH_SIZE`<br>`TVDB_patch_end_address:` |
| SGI IRIX | `.data`<br>`.align 3`<br>`.globl TVDB_patch_base_address`<br>`.globl TVDB_patch_end_address`<br>`TVDB_patch_base_address:`<br>`.space PATCH_SIZE`<br>`TVDB_patch_end_address:` |

Here's how you would use the static patch space assembler code:

1 Use an ASCII editor and place the assembler code into a file named **tvdb_patch_space.s**.

2 Replace the tag **PATCH_SPACE** with the decimal number of bytes you want. This value must be a multiple of 8.

3 Assemble the file into an object file using a command such as:

```
cc -c tvdb_patch_space.s
```

On SGI IRIX, also pass **–n32** or **–64** to create the correct object file type.

4 Link the resulting **tvdb_patch_space.o** into your program.

## Controlling Evaluation Points

The procedures for controlling evaluation points are identical to the procedures for controlling breakpoints and process barrier points. For more information, see:

■ "*Displaying the Action Points Window*" on page 210
■ "*Displaying and Controlling Action Points*" on page 211

# Using Watchpoints

TotalView lets you monitor the changes that occur to memory locations by creating a special kind of action point called a *Data Watchpoint*, or just *Watchpoint* for short. Watchpoints are most often used to find a statement in your program that is writing to a "stray" memory location. This can occur, for example, when memory is shared and another process or thread is writing to the same location, when writing off the end of an array, or when your program has a dangling pointer.

TotalView watchpoints are called "modify watchpoints" because TotalView only *triggers* a watchpoint when your program modifies a memory location. If a program writes a value into a location that is the same as what is already stored, TotalView does not trigger the watchpoint because its value did not change.

For example, if location 0x10000 has a value of zero and your program writes a zero into this location, TotalView does not trigger the watchpoint even though your program wrote data into the memory location. See "*Triggering Watchpoints*" on page 229 for more details on when watchpoints trigger.

TotalView also lets you create *conditional watchpoints*. A conditional watchpoint is similar to an evaluation point in that TotalView will evaluate an expression when the watchpoint triggers. You can use conditional watchpoints for a number of purposes. For example, you can use it to test if a value changes its sign—that is, it becomes positive or negative—or if a value moves above or below some threshold value.

## Architectures

The number of watchpoints, their size, and alignment restrictions differs from platform to platform. (This is because TotalView relies on the operating system and its hardware to implement data watchpoints.)

**NOTE** Watchpoints are not available on Alpha Linux and HP.

The following list describes constraints that are unique to each platform:

**Compaq Tru64** Watchpoints are implemented on Compaq Tru64 systems using a page protection scheme. Tru64 places no limitations on the number of watchpoints that you can create and there are no alignment or size constraints. However, watchpoints cannot overlap and you cannot create a watchpoint on an already write-protected page.

Because the page size is 8,192 bytes, using watchpoints can degrade performance if your program frequently writes to protected pages.

**IBM AIX** You can create one watchpoint on AIX 4.3.3.0-2 (AIX 4.3R) or later systems. (AIX 4.3R is available as APAR IY06844.) This watchpoint cannot be longer than 8-bytes and it must be aligned within an 8-byte boundary.

**IRIX6 MIPS** Watchpoints are implemented on IRIX 6.2 and later operating systems. These systems allow you to create about 100 watchpoints. There are no alignment or size constraints. However, watchpoints cannot overlap.

**Linux x86** You can create up to four watchpoints and each must be 1-, 2-, or 4-bytes in length and a memory address must be aligned for the byte length. That is, a 4-byte watchpoint must be aligned on a 4-byte address boundary, and a 2-byte watchpoint must be aligned on a 2-byte boundary, etc.

**Solaris SPARC/x86**

Watchpoints are implemented on Solaris 2.6 or later operating systems. These operating system allow you to create hundreds of watchpoints and there are no alignment or size constraints. However, watchpoints cannot overlap.

Typically, a debugging session does not use many watchpoints. In most cases, only one memory location at a time is being monitored. So, restrictions on the number of values you can watch are seldom an issue.

## Creating Watchpoints

Creating a watchpoint is a three-step process. The first step is to dive on a variable to display its Variable Window. With the cursor in the Variable Window, display the **Variable** popup menu, and select the **Watchpoint on Variable...** menu item. (If your platform does not support data watchpoints, this menu item is dimmed.)

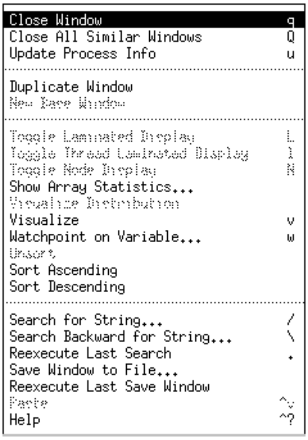



Figure 105: The Variable Menu

As an alternative, you could have typed "**w**" while the cursor is in the Variable Window.

**NOTE** Be careful that your cursor is focused on the Variable Window. If it is focused on the Process Window, typing "w" holds the process. If this occurs, you will need to manually release the process by typing "w" a second time.

After selecting the **Watchpoint on Variable** command. TotalView displays the dialog box shown in Figure 106.

The fields and controls in this window are as follows:

**Memory Address** The first (or lowest) memory address to watch. Depending on the platform, this address may need to be aligned to a multiple of the **Byte Size** field. For more information, see "*Architectures*" on page 223. If you edit the address of an existing watchpoint, TotalView alters

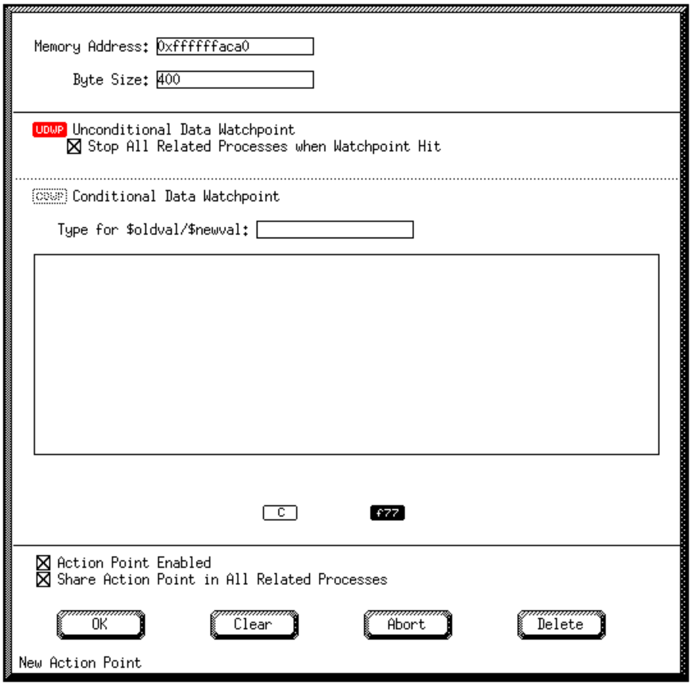



Figure 106: Watchpoint Options Dialog Box

the watchpoint so it will watch this new memory location and reassigns the watchpoint's action point ID.

**Byte Size**
The number of bytes being watched. Changing this value changes the number of bytes that will be watched. Your operating system may place constraints on the byte size. For more information, see "*Architectures*" on page 223.

**UDWP**
Unconditional Data Watchpoint. If this button is selected, TotalView will stop your program when the watchpoint triggers.

**Stop All Related Processes when Watchpoint Hit**
If selected, TotalView stops all members of the program group when the watchpoint triggers. Otherwise, only the process that reaches the watchpoint stops.

**CDWP** Conditional Data Watchpoint. If this button is selected, TotalView evaluates the expression when the watchpoint triggers.

**Type for $oldval/$newval**
This field lets you specify the data type of the **$oldval** and **$newval** built-in variables when you use them in a conditional watchpoint expression. This must be a scalar type, such as **int**, **integer**, **float**, **real**, or **char**. Aggregate types such as arrays and structures are not allowed.

If the size of the watched location matches the size of the data type entered here, the **$oldval** and **$newval** information is interpreted as the variable's type. If you watch an entire array, the watched location can be larger than the size of this type. For more information, see "*Conditional Watchpoints*" on page 230.

**(Evaluation area)** Enter the expression that TotalView will execute when the watchpoint triggers. The statements are written in the C or Fortran programming languages (as indicated by the **C** and **f77** buttons). For more information, see "*Conditional Watchpoints*" on page 230.

**Action Point Enabled**
If selected, indicates that the watchpoint is enabled. You can also toggle the enabled/disabled state by selecting the watchpoint in the **Action Points List** Pane.

**Share Action Point in All Related Processes**
If selected, enables and disables the watchpoint in all members of the share group. If this button is not selected, you must enable and disable the watchpoint in each share group member individually.

The controls at the bottom have their standard meanings:

**OK** Accepts all changes.

**Clear** Clears all fields and deselects all buttons.

**Abort** Cancels this dialog box without making changes.

**Delete** Deletes this watchpoint.

## Displaying Watchpoints using the Action Points Window

The Action Points Window displays a summary of the action points that are set in your program. To display this window, invoke the **STOP/BARR/EVAL/ ELOG** menu and select the **Open Action Points Window** command. The Action Points Window appears, as shown in Figure 107

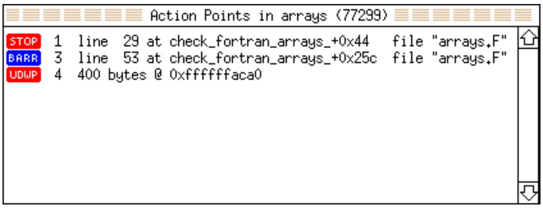



Figure 107: Action Points Window

The watchpoint entry, indicated by UDWP for Unconditional Data Watchpoint and CDWP for Conditional Data Watchpoint, displays the action point ID, the amount of memory being watched, and the location being watched.

If you dive into a watchpoint, TotalView displays the **Watchpoint Options** dialog box.

If you select a watchpoint, TotalView will toggle the enabled/disabled state of the watchpoint.

The list of action points displayed in the Action Points Window is the same as shown in the Action Points Pane in the Process Window. Diving into a watchpoint in this window also displays the **Watchpoint Options** dialog box.

## Watching Memory

A watchpoint tracks a memory location: it does not track a variable. This means that a watchpoint may not perform as you would expect it to when watching stack or automatic variables. For example, assume that you create a watchpoint to watch a variable in a subroutine. When control exits from the subroutine, the memory allocated on the stack for this subroutine is deallocated. At this time, TotalView is watching unallocated stack memory.

And, when the stack memory is reallocated to a new stack frame, the watchpoint will be triggered when that memory is modified.

Also, if a subroutine is reinvoked, it often executes using a different part of the stack. So, if the subroutine changes a variable within the subroutine, this change may not be seen because the variable is at a different memory location.

All of this means that in most circumstances, you can not place a watchpoint on a stack variable. If you need to watch a stack variable, you will need to create and delete the watchpoint each time your program invokes the subroutine.

**NOTE** **In some circumstances, a subroutine is always called from the same location. This means that its local variables will probably be in the same location, so it may be worth trying.**

If you place a watchpoint on a variable that is always invoked by reference (that is, the value of a variable is always accessed using a pointer to the variable), you can set a watchpoint on it because the memory locations used by the variable are not changing.

## Triggering Watchpoints

### The Program Counter after a Watchpoint Triggers

When a watchpoint triggers, the thread's program counter points to the instruction *following* the instruction that caused the watchpoint to trigger. If the memory store instruction is the last instruction in a source statement, the program counter will be pointing to the source line *following* the statement that triggered the watchpoint. (Breakpoints and watchpoints work differently. A breakpoint stops *before* an instruction executes. In contrast, a watchpoint stops *after* an instruction executes.)

### Multiple Watchpoints

If a program modifies more than one byte with one instruction (which is normally the case when storing a word), the watchpoint with the lowest memory location in the modified region is triggered. Although the program may be modifying locations monitored by other watchpoints, only the watchpoint for the lowest memory location is triggered. This situation occurs when your watchpoints are monitoring adjacent memory locations and a single store instruction modifies these adjacent locations.

For example, assume that you have two 1-byte watchpoints, one on location 0x10000 and the other on location 0x10001. Also assume that your program uses a single instruction to store a 2-byte value at locations 0x10000 and 0x10001. If the 2-byte storage operation modifies both bytes, the watchpoint for location 0x10000 triggers. The watchpoint for location 0x10001 does not and will not trigger at this time.

Here's a second example. Assume that you have a 4-byte integer that uses storage locations 0x10000 through 0x10003 and you set a watchpoint on this integer. If a process modifies location 0x10002, TotalView triggers the watchpoint. Now assume that you are watching two adjacent 4-byte integers that are stored in locations 0x10000 through 0x10007. If a process writes to locations 0x10003 and 0x10004 (that is, one byte within each), TotalView triggers the watchpoint associated with location 0x10003. The watchpoint associated with location 0x10004 does not trigger.

### Data Copies

TotalView keeps an internal copy of data in the watched memory locations for each process sharing the watchpoint. Consequently, if you create watchpoints that cover a large area of memory or if your program has a large number or processes, you will increase TotalView's virtual memory requirements. Further, TotalView refetches data for each memory location whenever the process or thread is continued. This can affect TotalView's performance.

## Conditional Watchpoints

If you associate an expression with a watchpoint (by selecting the **CDWP** button in the **Watchpoint Options** dialog box and typing in an expression), TotalView will evaluate the expression after the watchpoint triggers. The programming statements that you can use in this area are identical to those that you can use when creating an evaluation point, except that you are not allowed to call functions from a watchpoint expression.

The variables used in watchpoint expressions must be global. This is because the watchpoint can be triggered from any procedure or scope within your program.

Because memory locations are not scoped, the variable used in your expression must be globally accessible.

**NOTE** **Fortran does not have global variables. Consequently, you cannot directly refer to your program's variables.**

TotalView has two intrinsic variables that are used with conditional watchpoint expressions. These variables are:

| | |
|---|---|
| **$oldval** | The value of the memory locations before a change is made. |
| **$newval** | The value of the memory locations after a change is made. |

Here is an expression that uses these values:

```
if (iValue != 42 && iValue != 44) {
    iNewValue = $newval; iOldValue = $oldval; $stop;}
```

When the value **iValue** global variable is neither 42 nor 44, TotalView will store the new and old memory values in the **iNewValue** and **iOldValue** variables. These variables are defined in the program. (Storing the old and new values is a convenient way of letting you monitor the changes made by your program.)

Here is a condition that triggers a watchpoint when a memory location's value becomes negative:

```
if ($oldval >= 0 && $newval < 0) $stop
```

And here's a condition that triggers a watchpoint when the sign of the value in the memory location changes:

```
if ($newval * $oldval <= 0) $stop
```

Both of these examples require that you set the **Type for $oldval/$newval** field in the **Watchpoint Options** dialog box.

For more information on writing expressions, see "*Writing Code Fragments*" on page 235.

If a watchpoint has the same length as the **$oldval** or **$newval** data type, the value of these variables is apparent. However, if the data type is shorter than the length of the watch region, TotalView searches for the first changed location in the watched region and uses that location for **$oldval** and **$newval** variables. (It aligns data within the watched region based on the size of their type. For example, if their type is a 4-byte integer and byte 7 in the watched region changes, TotalView uses bytes 4 through 7 of the watchpoint when it assigns values to these variables.)

For example, suppose you are watching an array of 1000 ints called **must_be_positive** and you want to trigger a watchpoint as soon as one element becomes negative. You would declare the type for **$oldval** and **$newval** to be **int** and use the following condition:

```
if ($newval < 0) $stop;
```

When your program writes a new value to the array, TotalView triggers the watchpoint, sets the values of **$oldval** and **$newval**, and evaluates the expression. When **$newval** is negative, the **$stop** statement halts the process.

This can be a very powerful technique for range checking all the values written into an array. (Because of byte length restrictions, you can only use this technique on IRIX and Solaris.)

Conditional watchpoints are always interpreted by TotalView; they are never compiled. And, because interpreted watchpoints are single threaded within TotalView, every process or thread that writes to the watched location must wait for other instances of the watchpoint to finish executing. This can adversely affect performance.

# Saving Action Points in a File

You can save a program's action points into a file. TotalView will then use this information to reset these points when it is restarted. When you save action points, TotalView creates a file named *program*.**TVD.breakpoints**, where *program* is the name of your program.

**NOTE** Watchpoints are not saved.

To save action points, display the **STOP/BARR/EVAL/ELOG** menu and select the **Save All Action Points** command from the Process Window. TotalView places the action points file in the same directory as your program.

If you set "TOTALVIEW*AUTOSAVEBREAKPOINTS" on page 277, TotalView will automatically save action points to a file. Alternatively, starting TotalView with the **–sb** option (see "TotalView Command Syntax" described on page 299) also tells TotalView to save your breakpoints.

Once you create an action points file, TotalView automatically loads the file each time you invoke the debugger. TotalView uses the same search paths as it does to locate source files. If you prefer to suppress this behavior, you can set an X resource (see "TOTALVIEW*AUTOLOADBREAKPOINTS" on page 277) or use the **–nlb** option each time you start TotalView (see "*TotalView Command Syntax*" on page 299).

# Evaluating Expressions

TotalView lets you open a window for evaluating expressions in the context of a particular process and evaluate expressions in C, Fortran, or assembler.

**NOTE** **Not all platforms support the use of assembler constructs; see Appendix C "*Architectures*" on page 343 for details.**

To evaluate an expression:

1. Make sure that a process is created, running, or stopped in the Process Window.
2. Select the **Open Expression Window** command from the Process Window. An **Expression** Window appears.
3. Select the button (if it is not already selected) for the language in which you will write the code.

**4** Select the **Expression** box and enter a code fragment. For a description of the supported language constructs, see "*Writing Code Fragments*" on page 235.

The last statement in the code fragment can be a free-standing expression; you don't have to assign the expression's return value to a variable. Figure 108 shows a sample expression.

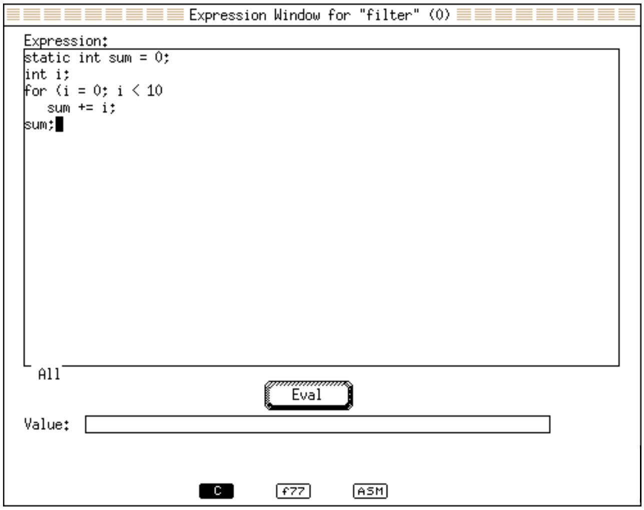



Figure 108: Sample Expression Window

**5** Select the **Eval** button. If TotalView finds an error, it places the cursor on the incorrect line and displays an error message. Otherwise, it interprets (or on some platforms, compiles and executes) the code, and displays the value of the last expression in the **Value** field.

While the code is being executed, you cannot modify anything in the window. TotalView also displays diagonal lines across the window, indicating that the window is temporarily inaccessible.

Since code fragments are evaluated in the context of the target process, stack variables are evaluated according to the currently selected stack frame. If the fragment reaches a breakpoint (or stops for any other reason), the **Expression** Window remains suspended. Assignment statements can affect the target process because they can change the value of a variable in the target process.

You can use the **Expression** Window in many different ways, but here are two examples:

- Expressions can contain loops, so you could use a **for** loop to search an array of structures for an entry set to a certain value. In this case, you use the loop index at which the value is found as the last expression in the **Expression** Window.
- Because you can call subroutines from the **Expression** Window, you can test and debug a single routine in your program without building a test program to call the routine.

Once you have selected and edited an expression in the window, you cannot use a keyboard equivalent (**q**) to exit from the window because the field editor is still active. To exit, display the menu and select the **Close Window** command or press Shift-Return.

# Writing Code Fragments

You can use code fragments in evaluation points and in the **Expression** Window. This section describes the intrinsic variables, built-in statements, and language constructs supported by TotalView.

## Intrinsic Variables

The TotalView expression system supports built-in variables that allow you to access special thread and process values. All of the variables are 32-bit integers, which is an **int** or a **long** on most platforms. The variables are not *lvalues*, so you cannot assign to them or take their addresses. Table 30 lists the intrinsic variable names and their meanings.

Table 30: **Intrinsic Variables**

| Name | Meaning |
|---|---|
| **$clid** | Returns the cluster ID. (Interpreted expressions only.) |
| **$duid** | Returns the TotalView-assigned Debugger Unique ID (DUID). (Interpreted expressions only.) |
| **$newval** | Returns the value just assigned to a watched memory location. (Watchpoints only) |

Table 30: Intrinsic Variables (cont.)

| Name | Meaning |
|---|---|
| **$nid** | Returns the node ID. (Interpreted expressions only.) |
| **$oldval** | Returns the value that existed in a watched memory location before a new value modified it. (Watchpoints only) |
| **$pid** | Returns the process ID. |
| **$processduid** | Returns the DUID of the process. (Interpreted expressions only.) |
| **$systid** | Returns the system-assigned thread ID. When referenced from a process, generates an error. |
| **$tid** | Returns the TotalView-assigned thread ID. When referenced from a process, generates an error. |

Intrinsic variables allow you to create thread specific breakpoints from the expression system. For example, the **$tid** intrinsic variable and the **$stop** built-in operation let you create thread specific breakpoint as follows:

```
if ($tid == 3)
    $stop;
```

This tells TotalView to stop the process only if thread 3 evaluated the expression. You can also create complex expressions using intrinsic variables. For example:

```
if ($pid != 34 && $tid > 7)
    printf ("Hello from %d.%d\n", $pid, $tid);
```

**NOTE** Using any of the following intrinsics means that the evaluation point is interpreted instead of compiled: $clid, $duid, $nid, $processduid, $systid, $tid, and $visualize. In addition, $pid forces interpretation on AIX.

## Built-In Statements

TotalView provides a set of built-in statements that you can use when writing code fragments. The statements are available in all languages, and are shown in the following table.

Table 31: **Built-In Statements Used in Expressions**

| Statement | Use |
|---|---|
| **$count** *expression*<br>**$countprocess** *expression* | Sets a process-level countdown breakpoint. When any thread in a process executes this statement for the number of times specified by *expression*, the process stops. The other processes in the program group continue to execute. |
| **$countall** *expression* | Sets a program-group-level countdown breakpoint. All processes in the program group stop when any process in the group executes this statement for the number of times specified by *expression*. |
| **$countthread** *expression* | Sets a thread-level countdown breakpoint. When any thread in a process executes, this statement the number of times specified by *expression*, it stops. Other threads in the process continue to execute.<br>If the target system does not support asynchronous stop, this statement is the same as **$countprocess**.<br>A thread evaluates *expression* when it executes **$count** for the first time, and this statement must evaluate to a positive integer. A thread re-evaluates **$count** only when it results in a breakpoint. After the breakpoint occurs, the debugger resets the process' internal counter for the breakpoint to the value of expression. The internal counter is stored in the process and shared by all threads in that process. |

Table 31: **Built-In Statements Used in Expressions (cont.)**

| Statement | Use |
|---|---|
| $hold<br>$holdprocess | Holds the current process. If all other processes in the group are already held in breakpoint state at this **Eval** point, then all are released. If other processes in the group are running, they continue to run. |
| $holdstopall<br>$holdprocessstopall | Exactly like **$hold**, except any processes in the group which are running are *stopped*. Note that the other processes in the group are not automatically held by this call—they are just stopped. |
| $holdthread | Freezes the current thread leaving other threads running. |
| $holdthreadstop<br>$holdthreadstopprocess | Exactly like **$holdthread** except it *stops* the *process*. The other processes in the group are left running. |
| $holdthreadstopall | Exactly like **$holdthreadstop** except it stops the entire group. |
| $stop<br>$stopprocess | Sets a process-level breakpoint. The process that executes this statement stops; other processes in the program group continue to execute. |
| $stopall | Sets a program-group-level breakpoint. All processes in the program group stop when any thread or process in the group executes this statement. |
| $stopthread | Sets a thread-level breakpoint. Although the thread that executes this statement stops, all other threads in the process continue to execute.<br><br>If the target system does not support asynchronous stop, this is the same as a **$stopprocess**. |

Table 31: **Built-In Statements Used in Expressions (cont.)**

| Statement | Use |
|---|---|
| **$visualize(***expression*[,*slice*]**)** | Visualizes the data specified by *expression* and modified by the optional *slice* value. *Expression* and *slice* must be written using the code fragment's language. The expression can be any valid expression that yields a data-set (after modification by *slice*) that can be visualized. *slice* is a quoted string containing a slice expression. For more information on how to use $visualize in an expression, see "*Visualizing Data in Expressions*" on page 253. |

## C Constructs Supported

When writing code fragments in C, keep these guidelines in mind.

- C-style (**/* comment */**) and C++-style (**// comment**) comments are permitted. For example:

```
// This code fragment creates a temporary patch
i = i + 2; /* Add two to i */
```

- You can omit semicolons when no ambiguity would result.
- Dollar signs ($) in identifiers are permitted.

### Data Types and Declarations

The following list describes the C data types and declarations that you can use:

- The data types that you can use are **char**, **short**, **int**, **float**, **double**, and pointers to any primitive type or any named type in the target program.
- Only simple declarations are permitted. The **struct**, **union**, and array declarations are not permitted.
- References to variables of any type in the target program are permitted.
- Unmodified variable declarations are considered local. References to them override references to similarly named global variables and other variables in the target program.

- (Compiled evaluation points only) The **global** declaration makes a variable available to other evaluation points and expression windows in the target process.
- (Compiled evaluation points only) The **extern** declaration references a global variable that was or will be defined elsewhere. If the global variable is not yet defined, TotalView displays a warning.
- Static variables are local and persist even after an evaluation point is evaluated.
- For static and global variables, expressions that initialize data as part of the variable declaration are performed only the first time the code fragment is evaluated. Local variables are initialized each time the code fragment is evaluated.

## Statements

The following list describes the C language statements that you can use.

- The statements that you can use are assignment, **break**, **continue**, **if**/**else** structures, **for**, **goto**, and **while**.
- You can use the **goto** statement to define and branch to symbolic labels. These labels are considered local to the window. As an extension, you can also refer to a line number in the target program. This line number refers to the *tag field* number of the source code line. Here is a **goto** statement that branches to source line number 432 of the target program:

```
goto 432;
```

- Although function calls are permitted, structures cannot be passed.
- Type casting is permitted.

All operators are permitted, with these limitations:

- The **?:** conditional operator is not supported.
- The **sizeof** operator can be used for variables, but not data types.
- The **(***type***)** operator cannot cast to fixed-dimension arrays using C cast syntax.

## Fortran Constructs Supported

When writing code fragments in Fortran, keep these guidelines in mind.

- Syntax is free-form. No column rules apply.
- One statement is allowed for each line; one line is allowed for each statement.

  **GOTO**, **GO TO**, **ENDIF**, and **END IF** are allowed; ELSEIF is not; use **ELSE IF**.
- Comment lines can be defined in several formats. For example:

```
C I=I+1
/*
I=I+1
J=J+1
ARRAY1(I,J)= I * J
*/
```

- The space character is significant and sometimes required. (Some Fortran 77 compilers ignore *all* space characters wherever they are coded.) For example:

| Valid | Invalid |
|---|---|
| DO 100 I=1,10 | DO100I=1,10 |
| CALL RINGBELL | CALL RING BELL |
| X .EQ. 1 | X.EQ.1 |

### Data Types and Declarations

The following is a list of data types and declarations that you can use within a Fortran expression.

- You can use the following data types: **INTEGER** (assumed to be long), **REAL**, **DOUBLE PRECISION**, and **COMPLEX**.
- Implied data types are not permitted.
- Only simple declarations are permitted. The **COMMON**, **BLOCK DATA**, **EQUIVALENCE**, **STRUCTURE**, **RECORD**, **UNION**, and array declarations are not permitted.
- References to variables of any type in the target program are permitted.

### Statements

The following list describes the Fortran language statements that you can use.

- You can use the following statements: assignment, **CALL** (to subroutines, functions, and all intrinsic functions except **CHARACTER** functions in the target program), **CONTINUE**, **DO**, **GOTO**, **IF** (including block **IF**, **ENDIF**, **ELSE**, and **ELSE IF**), and **RETURN** (but not alternate RETURN).
- As an extension to the **GOTO** statement, you can refer to a line number in the target program. This line number refers to the *tag field* number of the source code line. For example, this **GOTO** statement causes the program to branch to source line number 432 of the target program:

  ```
  GOTO $432;
  ```

  The dollar sign is required before the line number to distinguish the tag field number from a statement label.
- All expression operators are supported except **CHARACTER** operators and the logical operators **.EQV.**, **.NEQV.**, and **.XOR.**.
- Subroutine function and entry definitions are not permitted.
- Fortran 90 array syntax is not supported.
- Fortran 90 pointer assignment (the **=>** operator) is not supported.
- Calling Fortran 90 functions that require assumed shape array arguments is not supported.

## Writing Assembler Code

On Compaq Tru64 UNIX, RS/6000 IBM AIX, and SGI IRIX operating systems, TotalView lets you use assembler code in evaluation points, conditional breakpoints, and the Expression Window. However, if you want to use assembler constructs, you must enable compiled expressions. See "*Interpreted Versus Compiled Expressions*" on page 217 for instructions.

To indicate that an expression in the breakpoint or expression windows is an assembler expression, click on the **ASM** button in the Expression Window, as shown in the following figure.

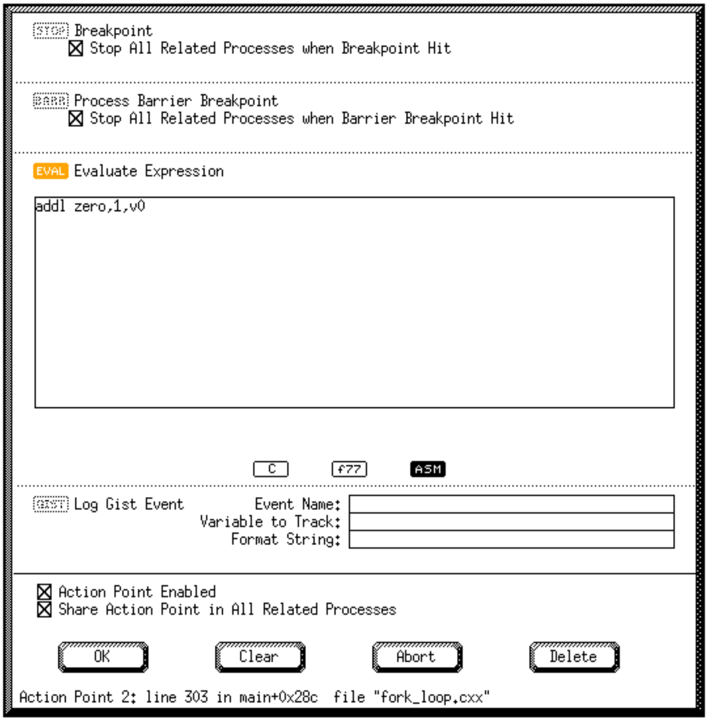



Figure 109: ASM Button in Expression Window

Assembler expressions are written in the TotalView Assembler Language. In this language, instructions are written in the target machine's native assembler language; the operators available to construct expressions in instruction operands and the set of available pseudo-operators, however, are the same on all machines.

The TotalView assembler accepts instructions using the same mnemonics recognized by the native assembler and recognizes the same names for registers that native assemblers recognize.

Some architectures provide extended mnemonics that do not correspond exactly with machine instructions and which represent important, special cases of instructions, or provide for assembling short, commonly used sequences of instructions. The TotalView assembler recognizes these mnemonics if:

- They assemble to exactly one instruction.
- The relationship between the operands of the extended mnemonics and the fields in the assembled instruction code is a simple one-to-one correspondence.

In TotalView Assembler Language, labels are indicated as *name*:, appearing at the beginning of a line. Labels may appear alone on a line. The symbols you can use include labels defined in the assembler expression and all program symbols.

The TotalView assembler operators are described in the following table:

Table 32: **TotalView Assembler Operators**

| Operators | Definition |
|---|---|
| + | Plus |
| – | Minus (also unary) |
| * | Times |
| # | Remainder |
| / | Quotient |
| & | Bitwise and |
| ^ | Bitwise xor |
| ! | Bitwise or not (also unary - bitwise not) |
| \| | Bitwise or |
| (*expr*) | Grouping |
| << | Left shift |
| >> | Right shift |
| "*text*" | Text string, 1-4 characters long, is right justified in a 32-bit word |
| hi16 (*expr*) | Low 16 bits of operand *expr* |
| hi32 (*expr*) | High 32 bits of operand *expr* |

Table 32: **TotalView Assembler Operators (cont.)**

| Operators | Definition |
|---|---|
| **lo16** (*expr*) | High 16 bits of operand *expr* |
| **lo32** (*expr*) | Low 32 bits of operand *expr* |

The TotalView Assembler pseudo-operations are listed in Table 33:

Table 33: **TotalView Assembler Pseudo Ops**

| Pseudo Ops | Definition |
|---|---|
| **$debug** [ **0** \| **1** ] | *Internal debugging option.*<br>With no operand, toggle debugging;<br>**0** => turn debugging off;<br>**1** => turn debugging on |
| **$hold**<br>**$holdprocess** | Hold the process |
| **$holdstopall**<br>**$holdprocessstopall** | Hold the process and stop the program group |
| **$holdthread** | Hold the thread |
| **$holdthreadstop**<br>**$holdthreadstopprocess** | Hold the thread and stop process |
| **$holdthreadstopall** | Hold the thread and stop the program group |
| **$long_branch** *expr* | Branch to location *expr*, using a single instruction in an architecture independent way, without requiring the use of any registers |
| **$ptree** | Internal debugging option.<br>Print assembler tree |
| **$stop**<br>**$stopprocess** | Stop the process |
| **$stopall** | Stop the program group |
| **$stopthread** | Stop the thread |
| *name=expr* | Same as **def** *name,expr* |
| **align** *expr* [**,** *expr* ] | Align location counter to an operand 1 alignment; use operand 2 (or zero) as the fill value for skipped bytes |
| **ascii** *string* | Same as *string* |
| **asciz** string | Zero terminated string |

Table 33: **TotalView Assembler Pseudo Ops (cont.)**

| Pseudo Ops | Definition |
|---|---|
| **bss** *name*,*size-expr*[,*expr*] | Define *name* to represent *size-expr* bytes of storage in the **bss** section with alignment optional *expr*; the default alignment depends on the size:<br>if *size-expr* >= 8 then 8 else<br>if *size-expr* >= 4 then 4 else<br>if *size-expr* >= 2 then 2 else 1 |
| **byte** *expr* [, *expr* ] ... | Place *expr* values into a series of bytes |
| **comm** *name*,*expr* | Define name to represent *expr* bytes of storage in the **bss** section; *name* is declared global; alignment is as in **bss** without an alignment argument |
| **data** | Assemble code into data section (data) |
| **def** *name*,*expr* | Define a symbol with *expr* as it's value |
| **double** *expr* [, *expr* ] ... | Place *expr* values into a series of doubles |
| **equiv** *name*,*name* | Make operand 1 be an abbreviation for operand 2 |
| **fill** *expr*, *expr*, *expr* | Fill storage with operand 1 objects of size operand 2, filled with value operand 3 |
| **float** *expr* [, *expr* ] ... | Place *expr* values into a series of floats |
| **global** *name* | Declare *name* as global |
| **half** *expr* [, *expr* ] ... | Place *expr* values into a series of 16 bit words |
| **lcomm** *name*,*expr*[,*expr*] | Identical to **bss** |
| **lsym** *name*,*expr* | Same as **def** *name*,*expr* but allows redefinition of a previously defined name |
| **org** *expr* [, *expr* ] | Set location counter to operand 1 use operand 2 (or zero) to fill skipped bytes |
| **quad** *expr* [, *expr* ] ... | Place *expr* values into a series of 64 bit words |
| **string** *string* | Place *string* into storage |
| **text** | Assemble code into text section (code) |
| **word** *expr* [, *expr* ] ... | Place *expr* values into a series of 32 bit words |
| **zero** *expr* | Fill *expr* bytes with zeros |

Chapter 9

# Visualizing Data

The TotalView Visualizer works with the TotalView debugger to create graphical images of your program's array data. In this chapter, you will learn:

- How the visualizer works
- Launching the Visualizer from TotalView
- Types of data that TotalView can visualize
- Visualizing data from the TotalView Variable Window
- Visualizing data using expressions
- What the Visualizer's windows do
- Changing settings from the Directory Window
- Methods of visualization
- Changing and manipulating the way data is displayed
- Launching the Visualizer from the command line
- Launching the Visualizer from a third party debugger
- Adapting third party visualizers to TotalView

The Visualizer is not available on all platforms.

## How the Visualizer Works

The Visualizer can be used in two ways: it can be launched from TotalView to visualize data as you debug your programs and it can be run from the command line to visualize data dumped to a file in a previous TotalView session.

Visualizing your program's data uses two interactions:

- You interact with TotalView to choose *what* you want to visualize and *when* it should make snapshots of your data.
- You interact with the visualizer to choose *how* you would like your data to be displayed.

The TotalView debugger handles the first of these interactions, extracting data and marshalling it into a standard format that it sends down a pipe. The Visualizer then reads the data from this pipe and displays it for analysis. The following figure shows this relationship.

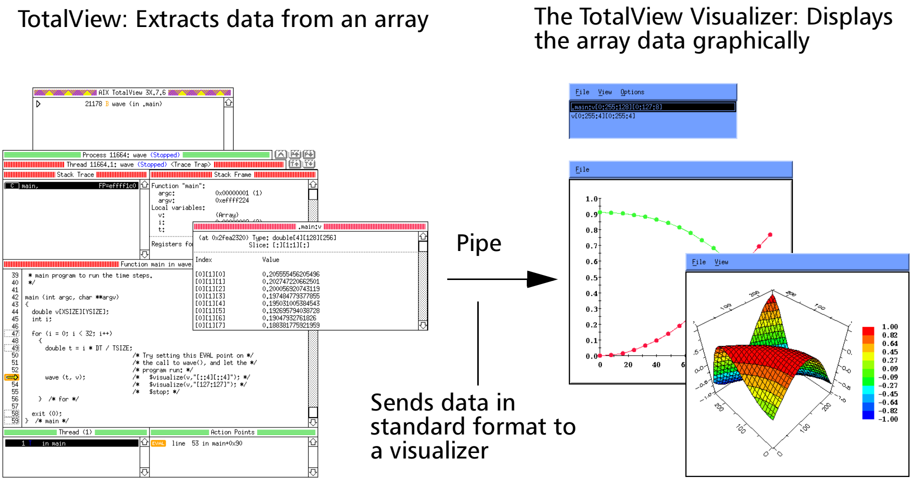



FIGURE 110: **TotalView Visualizer Connection**

You can send data directly from TotalView to the Visualizer while you are debugging your program. You can also send data from TotalView directly to a third party visualizer. Or, you can launch the TotalView Visualizer from the command line using data you have already saved to a file. Figure 111 shows these relationships.

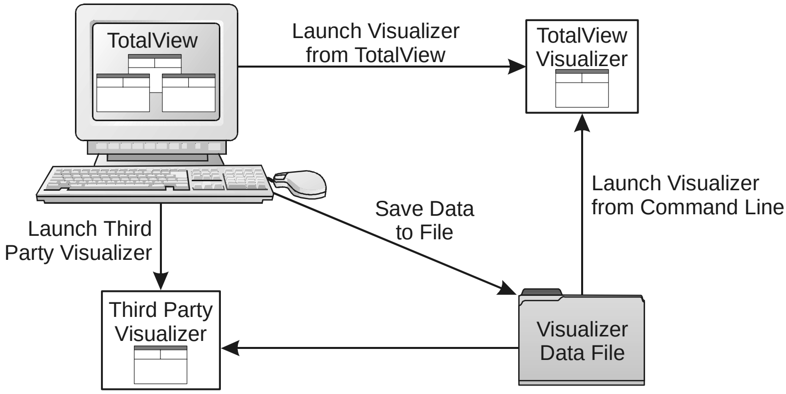



FIGURE 111: **TotalView Visualizer Relationships**

# Configuring TotalView to Launch the Visualizer

TotalView automatically launches the Visualizer when it is requested in a variable, breakpoint, or expression window. After TotalView launches the Visualizer, it pipes data to the Visualizer's standard input so you can visualize datasets as your program creates them.

If you disable visualization, TotalView silently ignores all attempts to use the Visualizer. This is useful when you want to execute some code containing evaluation points that do visualization, but do not want to individually disable all the evaluation points.

To change the Visualizer launch options interactively, select the **Visualizer Launch Window** from the Root Window. A dialog box appears, as shown in Figure 112. You can now tell the Visualizer to perform the following operations:

- Change the auto launch option. If you do not want it to launch the Visualizier automatically and disable visualization, clear the **TotalView Visualizer Auto Launch Enabled** checkbox.
- If the visualizer uses a customized command when it starts, enter it in the **Visualizer launch command** box.

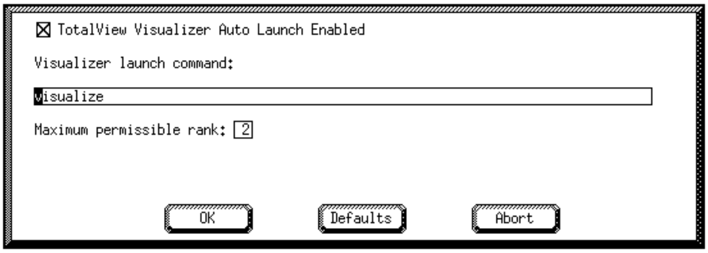



FIGURE 112: **The Visualizer Launch Window**

- Change the maximum permissible rank. Edit this value (the supported range is **1** through **16**) if you plan to save the data exported from the debugger or display it in a different visualizer.

  The maximum permissible rank (the default is **2**), ensures that data exported can be used in the TotalView Visualizer—the Visualizer displays only two dimensions of data. This limit does not apply to data saved in files, or to visualizers that can display more than two dimensions of data.
- Clicking on the **Defaults** button sets options to their defaults. This reverts to its standard default even if you have used an X resource to change it.

When you are done, click on the **OK** button. To abandon your edits, click on the **Abort** button.

If you disable visualization or change the visualizer launch string while a visualizer is running, TotalView closes the pipe to the visualizer. If you reenable visualization, TotalView launches a new Visualizer process the next time you visualize something.

You can change the shell command that TotalView uses to launch the visualizer by editing the Visualizer launch command. (You can even use this launch to command to run a different visualizer.) Or, you can save this information for viewing at another time. For example, you can save visualization information by entering the following command:

```
cat > your_file
```

Later, you can visualize this information using one of the following (equivalent) commands:

```
visualize –persist < your_file
visualize –file your_file
```

You can preset the visualizer launch options by setting X resources. For details, see Chapter 11 "X *Resources*" on page 275.

# Data Types that TotalView Can Visualize

The data selected for visualization is called a *dataset*. Each dataset is tagged with a numeric identifier that lets the Visualizer know whether it is seeing a new dataset or an update to an existing dataset. TotalView creates the identifier from the program, base address, and type of the data. This ensures that when you visualize the *same* data by different methods, the same set of images is updated. Note that stack variables at different recursion levels or call paths appear as separate images instead of updates to an existing image.

By default, TotalView restricts the type of data it can visualize to one and two dimensional arrays of character, integer, or floating point data. This data must be located in memory, and not in registers. You can visualize arrays with more dimensions by using an array slice expression to create a subarray with fewer dimensions. Figure 113 shows a three dimensional variable sliced into two dimensions by selecting a single index in the middle dimension.

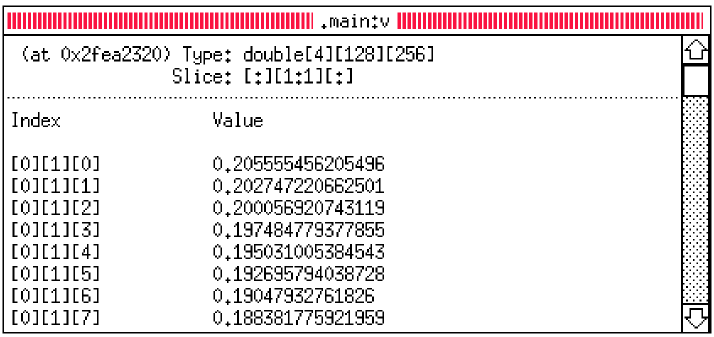



FIGURE 113: **A Three Dimensional Array Sliced to Two Dimensions**

# Visualizing Data from the Variable Window

The simplest way to visualize data is by using the Variable Window. (For details on the Variable Window, see Chapter 7 "*Examining and Changing Data*" on page 143.) Open a Variable Window on an array and stop program execution at the point where you want to visualize the array's values. Here is an example.

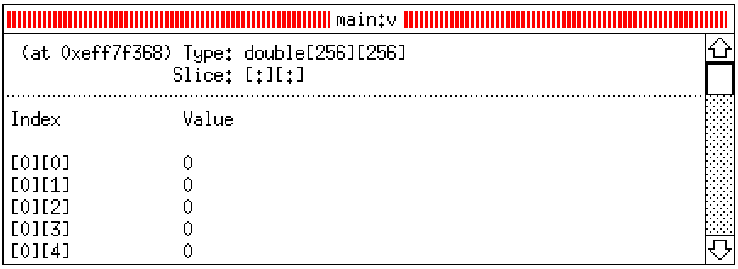



FIGURE 114: **Variable Window**

Editing the type and slice expressions fields lets you select the data you want visualized. You can display slices to limit the amount of data. (See "*Displaying Array Slices*" on page 167.) Limiting the amount increases the Visualizer's speed.

Launch the Visualizer program from the Variable Window by selecting the **Visualize** command from the **Variable** Window. The Visualizer will then create the initial Data Window display. If you reuse this command, TotalView send updated data values and the Visualizer updates its display.

You can visualize a Laminated Data Pane using the **Visualize** command. (See "*Visualizing a Laminated Data Pane*" on page 183.) The process or thread index forms one of the dimensions of the visualized data. This means that you can only visualize scalar or vector information. If you do not want the process or thread index as a dimension, use a non-laminated display.

Visualizer data displayed through a Variable Window is not automatically updated as you step through your program. You must explicitly request an update by reissuing the **Visualize** command while in a Variable Window.

# Visualizing Data in Expressions

The **$visualize** intrinsic (built-in) function lets you use TotalView's expression system to visualize data. This function lets you:

- Visualize several different variables from a single expression.
- Visualize variables in the Expression Evaluation Window.
- Visualize one or more variables from an evaluation point.

The syntax for the **$visualize** intrinsic is:

**$visualize (** *array* [, *slice_string* **])**

The *array* parameter is an expression naming the dataset being visualized. The optional *slice_string* parameter is a quoted string defining a constant slice expression that modifies the dataset named using the *array* parameter.

The following examples assume that your program has a two dimensional array called **my_array**.

**NOTE** In the following examples, notice that the array's dimension ordering differs.

TABLE 34: **$visualize examples for C and Fortran**

| C | Fortran |
|---|---|
| `$visualize(my_array);` | `$visualize (my_array)` |
| `$visualize (my_array,"[::2][10:15]")` | `$visualize (my_array,'(11:16,::2)')` |
| `$visualize (my_array,"[12][:]");` | `$visualize (my_array,'(:,13)')` |

The first example visualizes the entire array. The second example selects every second element in the array's major dimension; it also clips the minor dimension to all elements in the given (inclusive) range. The third example reduces the dataset to a single dimension by selecting one sub-array.

You may need a cast expression to let TotalView know what the dimensions of the variable being visualized are. For example, here is a procedure that passes a two dimensional array parameter that does not specify the extent of the major dimension.

```
void my_procedure (double my_array[][32])
{ /* procedure body */ }
```

The following cast expression is needed because the first dimension is not specified:

```
$visualize (*(double[32][32]*)my_array);
```

You can use **$visualize** in the expression window or by adding an expression to a breakpoint to create an evaluation point. But note that TotalView cannot compile an evaluation point or expression that contains **$visualize**. Instead, the TotalView debugger interprets these statements. See "*Defining Evaluation Points*" on page 213 for information about compiled and interpreted expressions.

Using **$visualize** in an expression window is a handy technique to refine an array and slice arguments or to update the Visualizer display of several arrays simultaneously.

## Visualizer Animation

Using **$visualize** in an evaluation point lets you animate the changes that occur in your data because the Visualizer will update the array's display every time TotalView reaches the evaluation point. This technique lets you create a visual animation of the array as the program executes.

# The TotalView Visualizer

The Visualizer has two types of windows:

- **A Directory Window**

  A single main window lists the datasets that you can visualize. You can use this window to set global options and to create views of your datasets.

- **Data Windows**

  The *Data Windows* contain images of the datasets. By interacting with a Data Window, you can change its appearance and set dataset viewing options. Using the Directory Window, you can open several Data Windows on a single dataset to get different views of the same data.

The following figure shows a Directory Window and two Data Windows. The left Data Window shows a graph view while the right window shows a surface view.

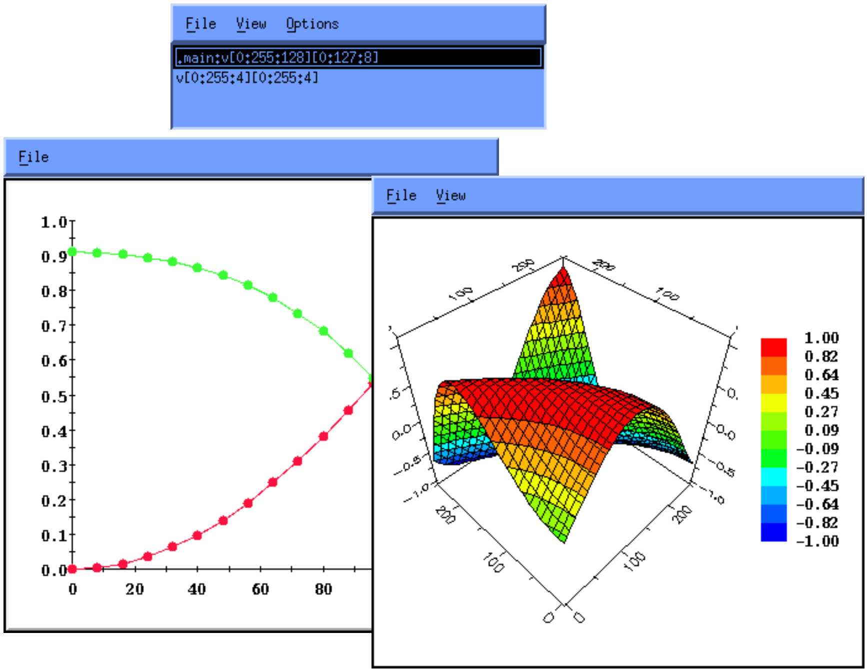



FIGURE 115: **Visualizer Windows**

## Directory Window

The Directory Window contains a list of the datasets you can display. For example:

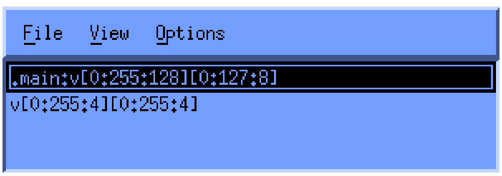



FIGURE 116: **Sample Visualizer Directory Window**

You can select a dataset by left-clicking on it and you can only select one dataset at a time. Right-clicking in the dataset list displays the **View** menu. From this menu, you can select **Graph** or **Surface** visualization. Whenever TotalView sends a new dataset, the Visualizer updates its list of datasets. To delete a dataset from the list, click on it, then display the **File** menu and select **Delete**.

You can automatically visualize the selected dataset by left-clicking in the dataset and pressing Return. You can also double-left-click in the dataset list to select and auto-visualize a dataset.

The following table shows the Directory Window's menubar commands.

TABLE 35: Directory Window Menu Commands

| Menu | Command | Meaning |
|---|---|---|
| File | Delete | Deletes the currently selected dataset. It removes the dataset from the dataset list and *destroys* any Data Windows displaying it |
| | Exit | Closes all windows and exits the **Visualizer** |
| View | Graph | Creates a new Graph Window; see "*Graph Data Window*" on page 259 for more detail |
| | Surface | Creates a new Surface Window; see "*Surface Data Window*" on page 261 for more detail |
| Options | Auto Visualize | This item is a toggle; when enabled, the **Visualizer** automatically visualizes new datasets as they are read |

## Data Windows

Data Windows display graphical images of your data. The following figure shows a surface view and a graph view. Every Data Window contains a menu bar and a drawing area. The Data Window title (which isn't shown in this figure) is its dataset identification.

The **File** menu on the menu bar is the same for all Data Windows. Any other items on the menu bar are specific to particular types of Data Window. The

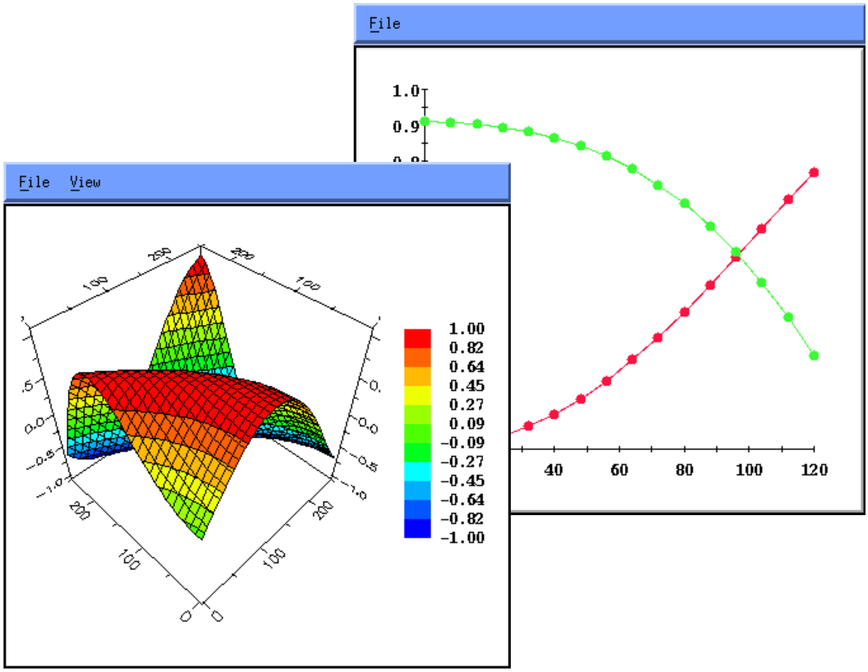



FIGURE 117: **Sample Visualizer Data Windows**

common Data Window menu commands are described in the following table.

TABLE 36: Data Window File Menu Commands

| Command | Meaning |
|---|---|
| Close | Closes the Data Window. |
| Delete | Deletes the Data Window's dataset from the dataset list. This also destroys any other Data Windows viewing the dataset. |
| Directory | Raises the Directory Window to the front of the desktop. If the Directory Window is minimized, it is restored. |
| New Base Window | Creates a new Data Window using the same visualization method and dataset as the current Data Window. |

TABLE 36: Data Window File Menu Commands (cont.)

| Command | Meaning |
|---|---|
| Options | Pops-up a window of viewing options. This window has a *control* area and an *action* area. The control area is specific to the Data Window. The action area contains three buttons as follows:<br>**OK**—Applies changes and removes the Options Window.<br>**Apply**—Applies the options settings, but leave the Options Window up.<br>**Cancel**—Closes the Options Window and discards any changes not yet applied. You can also cancel changes by closing the Options Window. |

The drawing area displays the image of your data. You can interact with the drawing area to alter the view of your data. For example, in the surface view, you can rotate the graph to view it from different angles. You can also get the value and indices of the dataset element nearest the cursor by left-clicking on it. A pop-up window displays the information. For details on this and other ways to manipulate a surface view, see Table 38 "*Surface Data Window Manipulations*" on page 264.

# Views of Data

Different types of datasets require different graphical views to display their data. For example, a graph is more suitable for displaying one dimensional datasets or two dimensional datasets where one of the dimensions has a small extent; however, a surface view is necessary for displaying a two dimensional dataset.

When the Visualizer is launched, one of the following actions will occur:

- If a Data Window is currently displaying the dataset, it is raised to the top of the desktop. If the window was minimized, it is restored.
- If dataset was previously visualized but no Data Window currently exists for it, the Visualizer creates a new Data Window using the most recent visualization method.

- If the dataset has never been visualized, the Visualizer chooses one a method, based on how well a given dataset matches an ideal dataset for each method.

The Visualizer can automatically choose a visualization method and create a new Data Window when it reads a new dataset. While the dataset is being updated, the Visualizer uses the method previously used. You can enable and disable this feature from the **Options** menu in the TotalView Visualizer Directory Window.

## Graph Data Window

The Graph Window displays a two dimensional graph of one or two dimensional datasets. If the dataset is two dimensional, the Visualizer displays multiple graphs. When you first create a Graph Window on a two dimensional dataset, the Visualizer uses the dimension with the larger number of elements for the X axis. It then draws a separate graph for each sub-array having the smaller number of elements. If you do not like this choice, you can transpose the data.

**NOTE** You probably do not want to use a graph to visualize two dimensional datasets with large extents in both dimensions as the display will be very cluttered.

You can display graphs with markers for each element of the dataset, with lines connecting dataset elements, or with both lines and markers as shown in Figure 118. See "*Displaying Graphs*" on page 260 for more details. Multiple graphs are displayed in different colors. The X axis of the graph is annotated with the indices of the long dimension. The Y axis shows you the data value.

You can scale and translate the graph, or pop up a window displaying the indices and values for individual dataset elements. See "*Manipulating Graphs*" on page 260 for details.

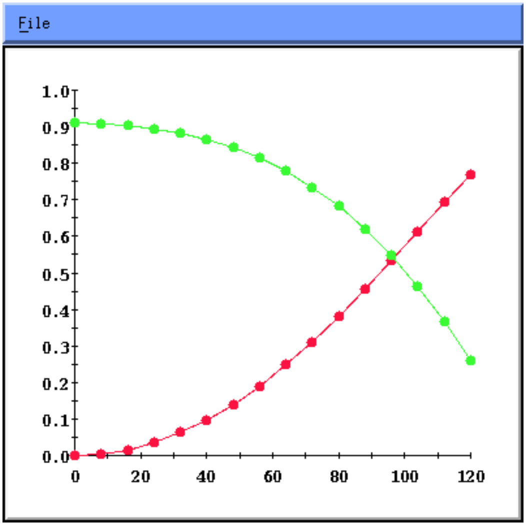



FIGURE 118: **Visualizer Graph Data Window**

## Displaying Graphs

The Graph Options dialog box, which is invoked by selecting the **Options** command on the **File** menu, lets you control how the Visualizer displays the graph, as is described in Table 37.

TABLE 37: Graph Data Window Options Dialog

| Toggle | Meaning |
|---|---|
| Lines | Toggles the display of lines connecting dataset elements |
| Points | Toggles the display of markers for each dataset element |
| Transpose | Toggles the choice of dimension to map onto the X axis of the graph for two dimensional datasets |

## Manipulating Graphs

You can manipulate the way the graph is displayed using the following actions:

**Scale** Press the Control key and hold down the middle mouse button. Move the mouse down to zoom in on the center of the drawing area, or up to zoom out.

**Translate** Press the Shift key and hold down the middle mouse button. Moving the mouse drags the graph.

**Zoom** Press the Control key and hold down the left mouse button. Drag the mouse button to create a rectangle that encloses an area. This area is then scaled to fit the drawing area.

**Reset View** Press **r** to reset the display to its initial state.

**Query** Hold down the left mouse button near a graph marker. A window pops up displaying the dataset element's indices and value.

Figure 119 shows a graph view of two dimensional random data created by selecting **Points** and deselecting **Lines** in the graph Data Window options dialog box.

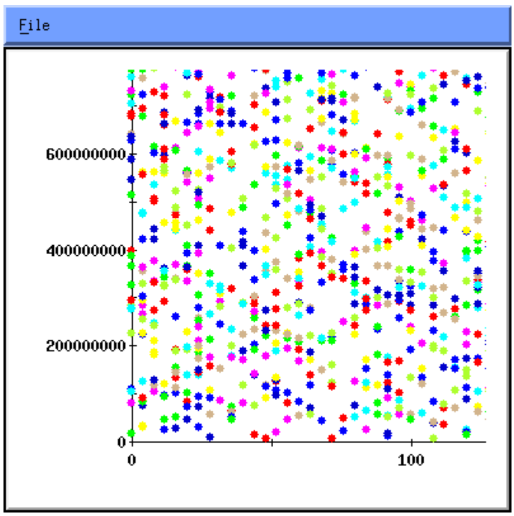



FIGURE 119: **Display of Random Data**

## Surface Data Window

The surface Data Window displays two-dimensional datasets as a surface in two or three dimensions. The dataset's array indices map to the first two dimensions (X and Y axes) of the display. Figure 120 shows a two dimensional map, where the dataset values are shown using only the **Zone** option.

(This demarcates ranges of element values.) For a zone map with contour lines, turn the **Zone** and **Contour** settings on and **Mesh** and **Shade** off.

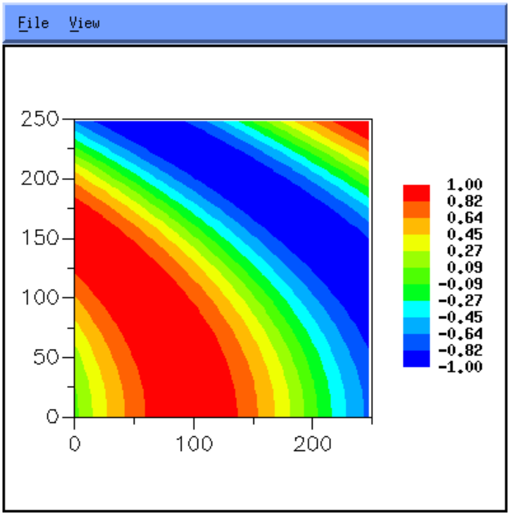



FIGURE 120: **Two Dimensional Surface Visualizer Data Display**

You can display random data by selecting only the **Zone** setting and turning **Mesh**, **Shade**, and **Contour** off. The display shows where the data is located and allows you to click on it to get the values of the various points.

Figure 121 shows a three dimensional surface that maps element values to the height (Z axis).

## Displaying Surface Data

The controls within the **Options** dialog box let you control the display of the surface data. In the Data Window, display the **File** menu and select the **Options** command. This dialog box has the following choices:

**Mesh** Toggles the *mesh* option. When this option is set, the surface is displayed in three dimensions, with the X-Y grid projected onto the surface. When neither this option nor the *shade* option are set, the surface is displayed in two dimensions (See Figure 120).

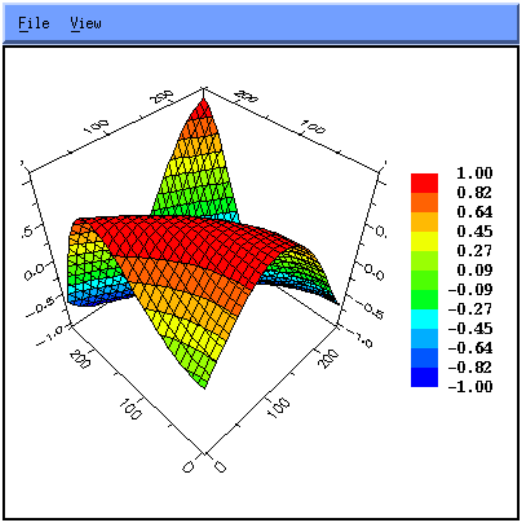



FIGURE 121: **Three Dimensional Surface Visualizer Data Display**

**Shade** Toggles the *shade* option. When this option is set, the surface is displayed in three dimensions and shaded either in a "flat" color to differentiate the top and bottom sides of the surface, or in colors corresponding to the value if the *zone* option is also set. When neither this option nor the *mesh* option are set, the surface is displayed in two dimensions. (See Figure 120.)

**Contour** Toggles the *contour* option. When this option is set, contour lines are displayed demarcating ranges of element values.

**Zone** Toggles the *zone* option. When this option is set, the surface is displayed in colors showing ranges of element values.

**Auto Reduce** Toggles the *auto reduce* option. When this option is set, the surface displayed is derived by averaging over neighboring elements in the original dataset. This speeds up visualization by reducing the resolution of the surface. Clear this option if you want to accurately visualize all dataset elements.

The **Auto Reduce** option allows you to choose between viewing all your data points—which takes longer to appear in the display—or viewing an averaging of data over a number of nearby points. The default for **Auto Reduce** is *on* so your display appears faster.

You can reset the viewing parameters to those used when the Visualizer first came up by selecting the. **Reset View** command contained within the **View** menu in the Data Window. This command restores all translation, rotation, and scaling. This resets the view of the surface to the initial state and enlarges the display slightly.

## Manipulating Surface Data

You can rotate a three dimensional surface to change the viewing angle so that you can see parts of the surface that are hidden or get a detailed view of part of the surface. When you click and hold the middle mouse button in the drawing area, then drag the mouse. The image changes to a wire-frame bounding box of the surface that moves with the mouse. You can rotate the view in two dimensions simultaneously, or select a single axis at a time to rotate. When you let go of the button, you can see the graph from the new, selected vantage point.

In addition to rotating the graph, you can manipulate it several other ways, as shown in Table 38. You can display the indices and values of individual dataset elements in a pop up window. You can control scaling and translating separately, or together with a zoom. You can query the values of individual elements. And you can reset the view to what it was when you started.

TABLE 38: Surface Data Window Manipulations

| Action | Description |
|---|---|
| Query | Hold down the left mouse button near the surface. A window pops up displaying the nearest dataset element's indices and value. |
| Reset View | Press **r** to reset translation and scaling. This does not reset the rotation. |
| Rotate | Hold down the middle mouse button and drag the mouse to freely rotate the surface. You can also press the **x**, **y**, or **z** keys to select a single axis of rotation. |

TABLE 38: Surface Data Window Manipulations (cont.)

| Action | Description |
|---|---|
| Scale | Press the Control key and hold down the middle mouse button. Move the mouse down to zoom in on the center of the drawing area, or up to zoom out. |
| Translate | Press the Shift key and hold down the middle mouse button. Moving the mouse drags the surface. |
| Zoom | Press the Control key and hold down the left mouse button. Drag the mouse button to create a rectangle that encloses the area of interest. The area is then translated and scaled to fit the drawing area. |

# Launching the Visualizer from Command Line

To start the Visualizer from the shell, use the following syntax:

**visualize [ -file** *filename* **| -persist ]**

where:

| | |
|---|---|
| **–file** *filename* | Reads data from *filename* instead of reading from standard input. |
| **–persist** | Continues to run after encountering an EOF on standard input. If you do not use this option, the Visualizer exits as soon as it reads all of the data from standard input. |

By default, the Visualizer reads its input data sets from its standard input stream and exits when it reads an EOF on standard input. When started by TotalView, the Visualizer normally reads its data from a pipe, ensuring that the Visualizer exits when TotalView does. If you want the Visualizer to continue to run after it exhausts all input from the standard input stream, invoke it using the **–persist** option.

If you want to read data from a file, invoke the Visualizer with the **–file** *filename* option. For example:

**visualize –file** *my_data_set_file*

The Visualizer reads all the datasets in the file. This means that the images you see are of the last versions of the datasets in the file.

The Visualizer supports the generic X toolkit command line options. For example, you can start the Visualizer with the Directory Window minimized by using the **–iconic** option. Your system manual page for the X server or the The X WINDOW SYSTEM USER'S GUIDE, by O'Reilly & Associates lists the generic X command line options in detail.

You can also customize the Visualizer by setting X resources in your resource files or on the command line with the **–xrm resource_setting** option. The available resources are described in Chapter 12 "*TotalView Command Syntax*" on page 299. Use of X resources to modify the default behavior of TotalView or the TotalView Visualizer is described in greater detail in Chapter 11 "X *Resources*" on page 275.

# Adapting a Third Party Visualizer

TotalView passes a stream of datasets to the Visualizer encoded in the format described later in this section. This means that you can use this data with other programs. Here are some things you should be aware of when using this data with other programs:

- TotalView and the Visualizer must be running on the same machine architectures; that is, TotalView assumes that word lengths, byte order, and floating-point representations are identical. While sufficient information in the dataset header exists to detect when this is not the case (with the exception of floating-point representation), no method for translating this information is supplied.
- TotalView transmits datasets down the pipe in a simple unidirectional flow. There is no handshaking protocol in the interface. This requires the Visualizer to be an *eager reader* on the pipe. If the Visualizer does not read eagerly, the pipe will back up and block TotalView.

The dataset format is described in the TotalView distribution in a header file named **include/visualize.h** in the TotalView installation directory. Each dataset is encoded with a fixed-length header followed by a stream of array elements. The header contains the following fields.

**vh_axis_order** Contains one of the constants **vis_ao_row_major** or **vis_ao_column_major**.

**vh_dims** Contains information on each dimension of the dataset. This includes a base, count and stride. Only the count is required to correctly parse the dataset. The base and stride only give information on the valid indices in the original data.

Note that all **VIS_MAXDIMS** of dimension information is included in the header, even if the data has fewer dimensions.

**vh_effective_rank**
Contains the number of dimensions that have an extent larger than 1.

**vh_id** Contains the dataset ID. Every dataset in a stream of datasets is numbered with a unique ID so that updates to a previous dataset can be distinguished from new datasets.

**vh_item_count** Contains the total number of elements to be expected.

**vh_item_length** Contains the length (in bytes) of single element of the array.

**vh_magic** Contains **VIS_MAGIC**, a symbolic constant to provide a check that this ***is*** a dataset header and that byte order is compatible.

**vh_title** Contains a plain text string of length **VIS_MAXSTRING** that annotates the dataset.

**vh_type** Contains one of the constants **vis_signed_int**, **vis_unsigned_int**, or **vis_float**.

**vh_version** Contains **VIS_VERSION**, a symbolic constant to provide a check that the reader understands the protocol.

Types in the dataset are encoded by a combination of the **vh_type** field and the **vh_item_length** field. This allows the format to handle arbitrary sizes of both signed and unsigned integers, and floating point numbers.

The **vis_float** constant corresponds to the default floating point format (usually, IEEE) of the target machine. The Visualizer does not handle values other than the default on machines that support more than one floating point format.

Although a three-byte integer is expressible in the Visualizer's dataset format, it is unlikely that the Visualizer will handle one. The Visualizer only handles data types that correspond to the C data types permitted on the machine where the Visualizer is running.

Similarly, the long double type varies significantly depending on the C compiler and target machine. Therefore, visualization of the long double type is unlikely to work if you run the Visualizer on a machine that is different from the one where you extracted the data.

In addition, you need to be aware of these data type differences if you write your own visualizer and plan to run it on a machine that is different from the one where you extract the data.

The data following the header is a stream of consecutive data values of the type indicated in the header. Consecutive data values in the input stream correspond to adjacent elements in **vh_dims[0]**.

You can verify that your reader's idea of the size of this type is consistent with TotalView by checking that the value of the **n_bytes** field of the header matches the product of the size of the type and the total number of array elements.

# Chapter 10

# Troubleshooting

**This chapter describes how to solve common problems that you might encounter while using TotalView.**

## Overview

This chapter discusses the following:

- Assembler is shown instead of source code
- Error creating new process
- Error launching process
- Fatal error: Checkout … failed
- Fatal error in TotalView
- Internal error in TotalView
- License manager doesn't operate correctly
- Out of memory error
- Pressing Ctrl-C in an xterm window causes TotalView to exit
- Program behaves differently under TotalView control: setuid issues
- Program behaves differently under TotalView control: SIGSTOP problems
- Program's symbols aren't shown
- Single stepping is slow or TotalView is slow to respond to breakpoints
- Source code doesn't appear in Source Code Pane
- TotalView can't find your source code
- TotalView server, tvdsvr, fails to start on a remote node

- When debugging HPF programs, HPF source code does not appear in the Process Window; only f77 code appears
- Windows do not appear or operate correctly
- X resources are not recognized

The TotalView Release Notes contains extensive information on known problems. There you will find information on configuring TotalView, required operating patches, and workarounds.

If you cannot solve a problem, please contact us. You will find our bug reporting form in the support area of our web site and in our Release Notes. Or, you can phone us at 1-800-856-3766 in the United States or (+1) 508-875-3030 worldwide.

# The Problems

### *Assembler is shown instead of source code*

Check to make sure that you compiled your program using **–g**.

### *Error creating new process*

- Increase the swap space on your machine. For details, see "*Swap Space*" on page 331.
- Increase the number of process slots in your system. See your operating system documentation for details.
- Check the **xterm** window to see if the **execve()** call failed, and if it did, set the PATH environment variable.
- Make sure that the **/proc** filesystem is mounted on your system. For details, see "*Mounting the /proc File System*" on page 330.

### *Error launching process*

- Run your program from the UNIX command line prompt to see if it will load and start executing. (If it won't start from the UNIX command line, TotalView will not be able to start it.)

  If it doesn't run, make sure your program is built for the machine on which you are debugging. Or, an **execv()** system call fails because the file does not have execute permission. Or, maybe you are trying to run a 64-bit application on a machine that only runs 32-bit applications.

- Check that all shared libraries needed by your application are accessible. For example, you may not have properly set the dynamic library runtime loader path (which is LD_LIBRARY_PATH most systems).
- Run your program from the UNIX command line prompt to see if it will load and start executing. If it begins executing, you can start TotalView, then attach to the executing program.
- TotalView cannot launch programs that are started by shell scripts. If it must be started by a shell script, you must manually start it then attach to it from within TotalView.

*Fatal error: Checkout ... failed*

- Check the value of the **LM_LICENSE_FILE** environment variable. Make sure the value ends with the string **license.dat**. The default location for this file is in the **flexlm-6.1** subdirectory within your TotalView installation directory.
- Make sure the TotalView license manager **lmgrd** is running on the license manager host machine. The name of this machine is listed in the **SERVER** line of your **license.dat** file. The default location for this daemon is in the **flexlm-6.1/*platform*/bin** subdirectory within your TotalView installation directory.
- Make sure that the **lmgrd** that is running matches the one which came with your TotalView distribution. That is, if you are running other software that uses the FLEX*lm* license manager or if you haven't upgraded an older version of FLEX*lm*, you might not be running the latest version.

*Fatal error in TotalView*

Report this problem. See "*Reporting Problems*" on page xvi.

*Internal error in TotalView*

Report this problem. See "*Reporting Problems*" on page xvi.

*Invalid license key*

Compare the format of your **license.dat** license key file with the one displayed in Chapter 2 of the TOTALVIEW INSTALLATION GUIDE. If you find stray characters in the file (for example "=3D"), use a text editor to remove them. After making these changes, stop the **lmgrd** license manager daemon and then restart it using the **toolworks_init** script.

### *License manager doesn't operate correctly*

Set the **LM_LICENSE_FILE** environment variable to the pathname of the TotalView license file. See the TOTALVIEW INSTALLATION GUIDE for details.

### *Out of memory error*

- Increase the swap space on your machine. For details, see "*Swap Space*" on page 331.
- Increase the data size limit in the C shell. Use the C shell's **limit** command, such as:

```
% limit datasize unlimited
```

### *Pressing Ctrl-C in an xterm window causes TotalView to exit*

Start TotalView using the **–ignore_control_c** command-line option.

### *Program behaves differently under TotalView control: setuid issues*

Make sure your program does not **setuid** or **exec** another program that does, for example, **rsh**. Normally, the operating system does not allow a debugger to debug a **setuid** executable nor allow a **setuid** system call while a program is being debugged. Often these operations fail silently. To debug **setuid** programs, login as the target UID before starting TotalView.

### *Program behaves differently under TotalView control: SIGSTOP problems*

TotalView uses the SIGSTOP signal to stop processes. On most UNIX systems, system calls can fail with *errno* set to **EINTR** when the process receives a SIGSTOP signal. You need to change your code so that it handles **EINTR** failures. For example:

```
do {
        n = read(fd,buf,nbytes);
} while (n < 0 && errno == EINTR);
```

When a system call is interrupted with a signal (for example, **errno == EINTR**), you need to retry it. This problem occurs because TotalView stops processes when it updates the displays. If the process is in a system call, the system call fails with **EINTR**.

For example, assume that your program has the following code fragment:

```
printf("creating scheduler thread...");
if (0 != (status = pthread_create(
                  &scheduler_thread, &detached_attr,
                  &scheduler_thread_wrapper, (void *)scheduler))) {
```

```
        error_func(ERR_LVL, __FILE__, __LINE__,
                "Pthread_create sScheduler, %d, %s",
                status, strerror(status));
}
```

You could restructure it to:

```
printf("creating scheduler thread...");
do {
status = pthread_create(
                &scheduler_thread, &detached_attr,
                &scheduler_thread_wrapper, (void *)scheduler);
    } while (0 != status && errno == EINTR);
if (0 != status) {
    error_func(ERR_LVL, __FILE__, __LINE__,
                "Pthread_create sScheduler, %d, %s",
                status, strerror(status));
}
```

### *Program’s symbols aren’t shown*

Check to make sure that you compiled your program using **–g**.

### *Single stepping is slow or TotalView is slow to respond to breakpoints*

- Close some of the Variable Windows that you have open.
- The Global Variables Window is open and has a large number of variables. Close the Global Variables Window.
- If you set a breakpoint in a source file that has not yet been referenced or if you single-step into one, TotalView must read the file’s symbol table. This can temporarily delay TotalView’s response.

### *Source code doesn’t appear in Source Code Pane*

- Set the search path for directories with the **Set Search Directory (d)** command in the Process Window.
- TotalView may be in the kernel or in a library routine for which source is not available.

### *TotalView can’t find your source code*

Set the search path for directories with the **Set Search Directory (d)** command in the Process Window.

### *TotalView server, tvdsvr, fails to start on a remote node*

Re-edit the server launch command field, click OK, and launch the server again. For information, see "*Starting the Debugger Server for Remote Debugging*" on page 55.

### *When debugging HPF programs, HPF source code does not appear in the Process Window; only f77 code appears*

When compiling HPF programs be sure to set the **–g** and **–Mtotalview** options when compiling and linking your programs.

### *Windows do not appear or operate correctly*

- Your **DISPLAY** *environment* variable is not set correctly.
- The resource "TOTALVIEW*USETRANSIENTFOR" on page 294 is not set correctly. Change it from **on** to **off**, or from **off** to **on**.
- Start Totalview with the **–grab** command-line option.
- Use the **xhost +** command to allow all hosts to access your display.

### *X resources are not recognized*

- Use the **xrdb** command (part of the X Window System) to display the current X resources:

  ```
  xrdb -query
  ```
- Use the **xrdb** command to load your X resources:

  ```
  xrdb -load $HOME/.Xdefaults
  ```
- Read the **xrdb** manual page for more information.

Chapter 11

# X Resources

**This chapter provides reference information about the X Window System resources that you can use to customize TotalView or the TotalView Visualizer. You can use these resources in your X resources files (such as .Xdefaults on UNIX systems or decw$sm_general.dat on VMS systems).**

For information on X resources files, refer to the X Window System documentation that came with your machine or the X WINDOW SYSTEM USER'S GUIDE, by O'Reilly & Associates (ISBN 1–56592–015–5).

On most UNIX systems, you load your X resources file using the **xrdb** command (part of the X Window System executables). For example:

```
xrdb –load $HOME/.Xdefaults
```

## TotalView X Resources

You can override some of the resources with command-line options for the **totalview** command, as described in Chapter 12 "*TotalView Command Syntax*" on page 299.

**NOTE** You can specify any of the following X resources on the command line using the "–*Xresource=value*" command line option. For example, to set totalview*stopAll to false, you could specify the –stopAll=false command line option. Note that the string "totalview*" is omitted from the command line

Window Locations: Values for the location of windows are expressed as:

```
=widthxheight+x+y
```

where *width* is the width of the window in pixels, *height* is the height of the window in pixels, *x* is the distance from the upper-left corner of the window to the left screen edge in pixels, and *y* is the distance from the upper-left corner of the window to the top screen edge in pixels. A value of **-1** for *x* or *y* indicates that the window should be centered in the screen with respect to the x-axis or y-axis. If desired, you can express *x* or *y* as negative numbers to indicate the distance from the lower-right corner of the window to the bottom screen edge or right screen edge instead of the distance from the upper-left corner. A value of zero (**0**) indicates that TotalView should use the default value. Also, you can supply just the size (*width* and *height*), and TotalView will use the default location (*x* and *y*) with it.

As an example, the expression **=0x0-1+20** uses the default width and height, centers the window horizontally, and places the window 20 pixels down from the top of the screen. The expression **=330x120+20-20** makes the window 330 pixels wide by 120 pixels high and places the window 20 pixels from the left edge of the screen and 20 pixels up from the bottom edge of the screen.

### totalview*arrowBackgroundColor: *color*

**Default: black**

Sets the background (outline) color of PC arrow to *color*.

### totalview*arrowForegroundColor: *color*

**Default: yellow2**

Sets the foreground (inner) color of PC arrow to *color*.

### totalview*askOnDlopen: {true | false}

If **true** (default), TotalView will ask you about stopping processes that dynamically load a new shared library using the **dlopen** or **load** (AIX only) system calls. If **false**, TotalView will never ask about stopping a process that dynamically loads a shared library. See "*Debugging Dynamically Loaded Libraries*" on page 338.

Override with: **–ask_on_dlopen** option (overrides **false**)
**–no_ask_on_dlopen** option (overrides **true**)

**totalview*autoLoadBreakpoints**: {**true** | **false**}

If **true** (default), automatically load action points from the file **filename.TVD.breakpoints**. If **false**, you use the **STOP/BARR/EVAL/ELOG → Load All Action Points** command in the Process Window to load action points.

Override with: **–lb** option (overrides **false**)
**–nlb** option (overrides **true**)

**totalview*autoRetraceAddresses**: {**on** | **off**}

If on (default), TotalView will retrace the sequence of dive operations performed in a Variable Window and recompute a new address for the variable. If off, TotalView does not retrace addresses.

**totalview*autoSaveBreakpoints**: {**true** | **false**}

If **false** (default), do not automatically save action points to an action points file when you exit. You use the **STOP/BARR/EVAL/ELOG → Save All Action Points** command in the Process Window to save action points.

Override with: **–sb** option (overrides **false**)
**–nsb** option (overrides **true**)

**totalview*backgroundColor**: *color*

**Default: white**

Sets the general background color to *color*.

**totalview*barrierForegroundColor**: *color*

**Default: blue**

Sets the color of the barrier point icon.

**totalview*barrierFontForegroundColor**: *color*

**Default: blue**

Sets the color of the font used to show the **H** and **Hold** indicators for held processes.

**totalview*barrierStopAll**: {**true** | **false**}

Same as
**totalview*processBarrierStopAllRelatedProcessesWhenBreakpointHit**.

**totalview*blindMouse**: {**on** | **off**}

If on (default), allow "*mouse ahead*," the queuing of mouse clicks (similar to typing ahead in a shell). If off, successive mouse clicks are ignored until TotalView responds to the first mouse click.

**totalview*breakFontForegroundColor**: *color*

**Default**: **orange**

Sets the color of "**B**" state to *color*.

**totalview*breakpointWindLocation**: =*widthxheight+x+y*

Specifies placement of the first Action Points Window.

| Default width | height | x | y |
|---|---|---|---|
| columns(70) | lines(12) | 335 | 10 |

**totalview*bulkLaunchBaseTimeout**: *n*

Sets the base timeout period when performing a bulk server launch to a value from 1 to 3600 (1 hour).

**totalview*bulkLaunchIncrTimeout**: *n*

Sets the incremental timeout period that TotalView waits for a process to launch during a bulk server launch. This value is from 1 to 360 (6 minutes).

**totalview*bulkLaunchEnabled**: {**true** | **false**}

If this resource is set to **true**, TotalView will auto-launch the TotalView Debugger Server (**tvdsvr**) when remote processes are launched

**totalview*bulkLaunchString**: *launch_string*

Defines the command that will be used to launch the TotalView Debugger Server (**tvdsvr**) when remote processes are created.

**totalview*buttonBackgroundColor**: *color*

Sets the button background color to *color*. Defaults to the background color.

**totalview*buttonForegroundColor**: *color*

Sets the button foreground color to *color*. Defaults to the foreground color.

**totalview*chaseMouse**: {**on** | **off**}

If on (default), display dialog boxes at the location of the mouse cursor. If off, display dialog boxes centered in the upper third of the screen.

Override with: **–chase** option (overrides **off**)
**–no_chase** option (overrides **on**)

**totalview*compilerVars**: {**true** | **false**}

Alpha Digital UNIX and SGI only. If **false** (default), TotalView does not show variables created by the Fortran compiler. If **true**, TotalView shows variables created by the Fortran compiler and the variables in the user's program.

Some Fortran compilers (Digital f90/f77, SGI 7.2 compilers) output debug information that describes variables that the compiler itself has invented for purposes such as passing the length of **character*(*)** variables. By default TotalView suppresses the display of these compiler generated variables; you can, however, setting **totalview*compilerVars** to **true** tells TotalView to display these variables. This could be useful if you are looking for a corruption of a run time descriptor or are writing a compiler.

Override with: **–compiler_vars** option (overrides **false)**
**–no_compiler_vars** option (overrides **true**)

**totalview*compileExpressions**: {**true** | **false**}

Alpha Digital UNIX and IBM AIX (default **true**), and MIPS IRIX (default **false**) platforms only. If **true**, TotalView enables compiled expressions. If **false**, TotalView disables compiled expressions and interprets them instead.

**totalview*conditionVariableInfoWindLocation**: =*width*x*height*+*x*+*y*

Specifies placement of the first Condition Variable Information Window.

| Default width | height | x | y |
|---|---|---|---|
| columns(75) | lines(15) | 360 | 300 |

### totalview*cTypeStrings: {**true** | **false**}

If **false** (default), use TotalView's type string extensions when displaying the type strings for arrays. If **true**, use C type string syntax when displaying arrays.

### totalview*dataWindLocation: =*width*x*height*+*x*+*y*

Specifies placement of the first Variable Window.

| Default width | height | x | y |
|---|---|---|---|
| columns(72) | max(205, lines(15)) | -80 | 320 |

### totalview*displayAssemblerSymbolically: {**on** | **off**}

If off (default), display assembler locations as hexadecimal addresses. If on, display assembler locations as "label+offset."

### totalview*dllIgnorePrefix: *prefix_list*

Sets the "DLL Don't Query on Load" prefix list to the space-separated list of prefixes specified in *prefix_list*. If "TOTALVIEW*ASKONDLOPEN" ON PAGE 276 is set to **true**, and the suffix of the library being loaded does *not* match a suffix on the "DLL Do Query on Load" suffix list, and if one or more of the prefixes in this list match the name of the library being loaded, then TotalView will not ask you if you would like to stop the process. For more information and the list of default prefixes by platform, see "*Debugging Dynamically Loaded Libraries*" on page 338.

### totalview*dllStopSuffix: *suffix_list*

Sets the "DLL Do Query on Load" suffix list to the space-separated list of suffixes specified in *suffix_list*. If "TOTALVIEW*ASKONDLOPEN" ON PAGE 276 is set to **true**, and if one or more of the suffixes in this list match the name of the library being loaded, then TotalView will ask you if you would like to stop the process. The *suffix_list* is empty by default. See "*Debugging Dynamically Loaded Libraries*" on page 338.

**totalview*DPVMDebugging**: {**true** | **false**}

*Digital* UNIX *only*.

If **false** (default), disables support for debugging the Digital UNIX implementation of Parallel Virtual Machine (DPVM) applications. If **true**, enables support for debugging DPVM applications.

Override with: **–dpvm** option (overrides **false**)
**–no_dpvm** option (overrides **true**)

**totalview*editorLaunchString**: *command_string*

**Default**: **xterm –e %E +%N %S**

Sets the editor launch command string to the specified value. Refer to "*Changing the Editor Launch String*" on page 121 for more information on the format of *command_string*.

**totalview*errorFontForegroundColor**: *color*

**Default**: **red**

Sets the color of "**E**", "**Z**", and "**?**" states to *color*.

**totalview*evalForegroundColor**: *color*

**Default**: **orange**

Sets the color of the **EVAL** icon to *color*.

**totalview*evalWindLocation**: =*width*x*height*+*x*+*y*

Specifies placement of the first Expression Evaluation Window.

| Default width | height | x | y |
|---|---|---|---|
| columns(83) | lines(30) + 2 | -1 | 10 |

**totalview*eventLogWindLocation**: =*width*x*height*+*x*+*y*

Specifies placement of the Event Log Window.

| Default width | height | x | y |
|---|---|---|---|
| columns(75) | lines(20) | -75 | -50 |

**totalview*font**: *fontname*

**Default**: **fixed**

Specifies the font used by the TotalView debugger. Use the X Windows supplied application **xlsfonts** to list the names of available fonts.

**totalview*foregroundColor**: *color*

**Default**: **black**

Sets the general foreground color (that is, the text color) to *color*.

**totalview*frameOffsetX**: *n*

**Default**: **0**

Sets the horizontal placement offset between windows of the same type, as TotalView places them on the screen. This value is *added* to the default value used by TotalView. If you are using TotalView title bars, use the default.

**totalview*frameOffsetY**: *n*

**Default**: **0**

Sets the vertical placement offset between windows of the same type, as TotalView places them on the screen. This value is *added* to the default value used by TotalView. If you are using TotalView title bars, use the default.

**totalview*globalsWindLocation**: =*width*x*height*+*x*+*y*

Specifies placement of the Global Variables Window.

| Default width | height | x | y |
|---|---|---|---|
| columns(62) | max(205, lines(15)) | -80 | 10 |

**totalview*globalTypenames**: {**true** \| **false**}

If **true** (default), specifies that TotalView can assume that type names are globally unique within a program and that all type definitions with the same name are identical. In C++, the standard asserts that this must be **true** for standard conforming code.

If this option is **true**, TotalView attempts to replace an opaque type (**struct foo *p;**) declared in one module, with an identically named defined type (**struct foo { … };**) in a different module.

If TotalView has read the symbols for the module containing the non-opaque type definition, then when displaying variables declared with the opaque type, TotalView will automatically display the variable using the non-opaque type definition.

If **false**, TotalView will not assume that type names are globally unique within a program. You should specify this option if your code has different definitions of the same named type, since otherwise TotalView is likely to pick the wrong definition to substitute for an opaque type.

Override with: **–global_types** option (overrides **false**)
**–no_global_types** option (overrides **true**)

### totalview*grabMouse: {on | off}

If off (default), do not force keyboard input to dialog boxes. If you're running TotalView with a window manager that is operating in "click-to-type" mode, you should set this resource to "on" or use the **–grab** command-line option.

### totalview*helpWindLocation: =*width*x*height*+*x*+*y*

Specifies placement of the help window.

| Default width | height | x | y |
|---|---|---|---|
| min(screen_width - 10, columns(84)) | min(screen_height - 20, 606) | -1 | -20 |

### totalview*hpf: {true | false}

If **true** (default, if HPF debugging has been licensed), enables debugging at the HPF source level.

Setting this X resource to **false,** causes TotalView to ignore **.stx** and **.stb** files, and therefore to debug HFP code at the intermediate (Fortran 77) level.

Override with: **–hpf** option (overrides **false**)
**–no_hpf** option (overrides **true**)

### totalview*hpfNode: {true | false}

If **false** (default), the node on which an HPF distributed array element resides is not displayed in the Process Window.

The node display can be toggled in each Variable Window using the **Toggle Node Display** option in the Process Window menu.

Override with: **–hpf_node** option (overrides **false**)
**–no_hpf_node** option (overrides **true**)

**totalview*inverseVideo**: {**true** | **false**}

If **true**, enables inverse video display. If **false** (default), disables inverse video display.

**totalview*kccClasses**: {**true** | **false**}

If set to **true**, (default) TotalView will convert structure definitions output by the KCC compiler into classes that show base classes, and virtual base classes in the same way as other C++ compilers. When set to **false**, the debugger will not convert structure definitions output by the KCC compiler into classes. Virtual bases will show up as pointers, rather than the data.

Unfortunately, the conversion has to be done by textual matching of the names given to structure members, so can it be confused if you have structure component names that look to TotalView like KCC processed classes. However, the conversion is never performed unless TotalView believes that the code was compiled with KCC, because TotalView has seen one of the tag strings that KCC outputs, or because the user has asked for the KCC name demangler to be used. Also all of the recognized structure component names start with "__", and, according to the C standard, user code should not contain names with this prefix.

Note that under some circumstances it is not possible to convert the original type names because there is no available type definition. For example, it may not be possible to convert "**struct __SO_foo**" to "**struct foo**", so in this case the "**__SO_foo**" type will be shown. This is only a cosmetic problem. (The "**__SO__**" prefix denotes a type definition for the non-virtual components of a class with virtual bases).

Since KCC outputs no information on the accessibility of base classes ("private", "protected", "public"), TotalView is unable to provide this information.

**totalview*mainHSplit**: *n*

Same as **totalview*mainHSplit1**.

### totalview*mainHSplit1: *n*

**Default**: (window_height/3)

Controls the height of the Stack Trace, Stack Frame, and Source Code Panes in the Process Window. *n* specifies the pixel location of the top of the Source Code Pane.

### totalview*mainHSplit2: *n*

Controls the height of the Source Code Pane, Thread List, and Action Point list in the Process Window. *n* specifies the pixel location of the top of the Thread List and Action Point List Panes.

Default: A function of *window_height*

TotalView tries to give 5 lines in the Thread List and Action Point list Panes, and the remainder, at least 20 lines, to the Source Code Pane. If it cannot give the Source Code Pane at least 20 lines, it shrinks the Thread List and Action Point List Panes to zero.

### totalview*mainVSplit: *n*

Same as **totalview*mainVSplit1**.

### totalview*mainVSplit1: *n*

**Default**: (window_width/2) – 20

Controls the location of the partition between the Stack Trace and Stack Frame Panes in the Process Window. A value of **–1** centers the partition.

### totalview*mainVSplit2: *n*

**Default**: (window_width/2) – 20

Controls the location of the partition between the Thread List and Action Point List Panes in the Process Window. A value of **–1** centers the partition.

### totalview*mainWindLocation: =*width*x*height*+*x*+*y*

Specifies placement of the first main Process Window.

| Default width | height | x | y |
|---|---|---|---|
| min(columns(94), screen_width - 5) | max(456, lines(45)) | 10 | -150 |

### totalview*menuArrowForegroundColor: *color*

Default: **blue** or **green**

Sets the menu arrow color to *color*.

### totalview*menuCache: {on | off}

If off (default), disables menu caching. Not all X servers support menu caching.

**NOTE** If your X server doesn't and you have menu caching enabled (on), TotalView menus appear blank the second and subsequent times you display them.

### totalview*messageStateWindLocation: =*width*x*height*+*x*+*y*

Specifies the placement of the first Message State Window.

| Default width | height | x | y |
|---|---|---|---|
| columns(72) | max(205, lines(15)) | -80 | 330 |

### totalview*modulesWindLocation: =*width*x*height*+*x*+*y*

Specifies the placement of the first Modules Window.

| Default width | height | x | y |
|---|---|---|---|
| columns(62) | max(205, lines(15)) | -75 | 15 |

### totalview*mouseCursorBackgroundColor: *color*

Default: **white** or **black**

Sets the mouse cursor background (mask) color to *color*.

### totalview*mouseCursorForegroundColor: *color*

Default: red

Sets the mouse cursor foreground (inner) color to *color*.

### totalview*multForegroundColor: *color*

Default: purple

Sets the color of **MULT** icon to *color*.

### totalview*mutexWindLocation: =*widthxheight+x+y*

Specifies placement of the first Mutex Information Window.

| Default width | height | x | y |
|---|---|---|---|
| columns(75) | lines(15) | 350 | 300 |

### totalview*overrideRedirect: {on | off}

If off (default), do not create TotalView windows using the **override_redirect** attribute. If on, use the **override_redirect** attribute, which does not give the X window manager a chance to intercept requests.

### totalview*ownTitles: {on | off}

If on (default), place title bars on TotalView windows. If your window manager is a reparenting one (places its own title bars on windows), turn off this resource.

### totalview*patchAreaAddress: *address*

Allocate the patch space dynamically at the given *address*. See "*Allocating Patch Space for Compiled Expressions*" on page 220.

### totalview*patchAreaLength: *length*

Set the length of the dynamically allocated patch space to the specified *length*. See "*Allocating Patch Space for Compiled Expressions*" on page 220.

### totalview*popAtBreakpoint: {on | off}

If on, sets the **Open (or raise) process window at breakpoint** checkbox to be selected by default. If off (default), sets that checkbox to be deselected by default. See "*Handling Signals*" on page 41.

Override with: **–pop_at_breakpoint** option (overrides **off**)
**–no_pop_at_breakpoint** option (overrides **on**)

### totalview*popOnError: {on | off}

If on (default), sets the **Open (or raise) process window on error** checkbox to be selected by default. If off, sets that checkbox to be deselected by default. "*Handling Signals*" on page 41.

Override with: **–pop_on_error** option (overrides **off**)
**–no_pop_on_error** option (overrides **on**)

**totalview*processBarrierStopAll**: {**true** | **false**}

Same as **totalview*processBarrierStopAllRelatedProcessesWhen-BreakpointHit**.

**totalview*processBarrierStopAllRelatedProcessesWhenBreakpointHit**: {**true** | **false**}

If **true** (default), the default setting for process barrier breakpoints stops all related processes. If **false**, the default setting for process barrier breakpoints does *not* stop all related processes. See "*Process Barrier Breakpoints*" on page 206.

**totalview*pullRightMenus**: {**on** | **off**}

If off (default), use walking menus. If on, use pull-right menus.

**totalview*pvmDebugging**: {**true** | **false**}

If **false** (default), disables support for debugging the ORNL implementation of Parallel Virtual Machine (PVM) applications. If **true**, enables support for debugging PVM applications.

Override with: **–pvm** option (overrides **false**)
**–nopvm** option (overrides **true**)

**totalview*rootWindLocation**: =*width*x*height*+*x*+*y*

Specifies placement of the Root Window.

| Default width | height | x | y |
|---|---|---|---|
| min(screen_width - 10, columns(60)) | max(150, lines(12)) | 10 | 10 |

**totalview*runningFontForegroundColor**: *color*

**Default: green**

Sets the color of "**R**", "**S**", "**M**", and "**I**" states to *color*.

### totalview*scrollLineSpeed: *n*

**Default**: 40

Specifies the maximum number of lines per second that TotalView scrolls when you click on arrows at the top and bottom of the scroll bars. To have TotalView scroll as fast as possible, set *n* to **0**.

### totalview*scrollPageSpeed: *n*

**Default**: 5

Specifies the maximum number of pages per second that TotalView scrolls when you click above or below the elevator box inside the scroll bars. To have TotalView scroll as fast as possible, set *n* to **0**.

### totalview*searchCaseSensitive: {**on** | **off**}

If off (default), searching for strings is not case-sensitive. If on, searches are case- sensitive.

### totalview*searchPath: *dir*1[,*dir*2,...]

Specifies a list of directories for the debugger to search when looking for source and object files. This resource serves the same purpose as the **Set Search Directory** command in the Process Window (see "*Setting Search Paths*" on page 44). If you use multiple lines, place a backslash (\) at the end of each line, except for the last line.

### totalview*serverLaunchEnabled: {**true** | **false**}

If **true** (default), TotalView automatically launches the TotalView Debugger Server (**tvdsvr**) when you start to debug a remote process.

### totalview*serverLaunchString: *command_string*

Specifies the command string that TotalView uses to automatically launch the TotalView Debugger Server (**tvdsvr**) when you start to debug a remote process. *command_string* is executed by **/bin/sh**. By default, TotalView uses the **rsh** command to start the server, but you can use any other command that can invoke **tvdsvr** on a remote host. If you have no command available for invoking a remote process, you can't automatically launch the server; therefore, you should set **totalview*serverLaunchEnabled** to **false**.

Default: `%C %R -n “cd %D && tvdsvr -callback %L \`
`-set_pw %P -verbosity %V“`

**totalview*serverLaunchTimeout:** *n*

Default: 30

Specifies the number of seconds that TotalView waits to hear back from the TotalView Debugger Server (**tvdsvr**) that it launched successfully. The number of seconds must be between **1** and **3600** (1 hour).

**totalview*shareActionPoint:** {**true** | **false**}

Same as **totalview*shareActionPointInAllRelatedProcesses**.

**totalview*shareActionPointInAllRelatedProcesses:** {**true** | **false**}

If **true** (default), the default setting for action points will be to share them in all related processes. If **false**, the default setting for action points will be to *not* share them in all related processes. See “B*reakpoints for* M*ultiple* P*rocesses*” on page 203.

**totalview*signalHandlingMode:** *action_list*

Modifies the way in which TotalView handles signals. An *action_list* consists of a list of *signal_action* descriptions, separated by spaces:

*signal_action*[*signal_action*] ...

A signal_action description consists of an action, an equal sign (=), and a list of signals:

*action*=*signal_list*

An *action* can be one of the following: **Error**, **Stop**, **Resend**, or **Discard**. For more information on the meaning of each action, refer to “H*andling* S*ignals*” on page 41.

A *signal_list* is a list of one or more signal specifiers, separated by commas:

*signal_specifier*[**,***signal_specifier*] ...

A *signal_specifier* can be a signal name (such as **SIGSEGV**), a signal number (such as **11**), or a star (*****), which specifies all signals. We recommend using the signal name rather than the number because number assignments vary across UNIX versions.

The following rules apply when specifying an *action_list*:

- If you specify an action for a signal in an *action_list*, TotalView changes the default action for that signal.
- If you do not specify a signal in the *action_list*, TotalView does not change its default action for the signal.
- If you specify a signal that does not exist for the platform, TotalView ignores it.
- If you specify an action for a signal twice, TotalView uses the last action specified. In other words, TotalView applies the actions from left to right.

If you need to revert the settings for signal handling to TotalView's built-in defaults, use the **Defaults** button in the **Set Signal Handling Mode** dialog box.

For example, to set the default action for the **SIGTERM** signal to **Resend**, you specify the following action list:

```
"Resend=SIGTERM"
```

As another example, to set the action for **SIGSEGV** and **SIGBUS** to **Error**, the action for **SIGHUP** and **SIGTERM** to **Resend**, and all remaining signals to **Stop**, you specify the following action list:

```
"Stop=* Error=SIGSEGV,SIGBUS Resend=SIGHUP,SIGTERM"
```

This action list shows how TotalView applies the actions from left to right. The action list first sets the action for all signals to **Stop**. Then, the action list changes the action for **SIGSEGV** and **SIGBUS** from **Stop** to **Error** and the action for **SIGHUP** and **SIGTERM** from **Stop** to **Resend**.

### totalview*sourcePaneTabWidth: *n*

**Default: 8**

Sets the width of the tab character that is displayed in the Source Pane. For example, if your source file uses a tab width of 4, set *n* to **4**.

### totalview*spellCorrection: {verbose | brief | none}

When you use the **Function or File...** or **Variable...** commands in the Process Window or edit a type string in a Variable Window, the debugger checks the spelling of your entries. By default (**verbose**), the debugger displays a dialog box before it corrects spelling. You can set this resource to **brief** to run the spelling corrector silently. (The debugger makes the spelling correction without displaying it in a dialog box first.) You can also set this resource to **none** to disable the spelling corrector.

### totalview*stopAll: {true | false}

Same as **totalview*stopAllRelatedProcessesWhenBreakpointHit**.

### totalview*stopAllRelatedProcessesWhenBreakpointHit: {true | false}

If **true** (default), the default setting for breakpoints will stop all related processes. If **false**, the default setting for breakpoints will *not* stop all related processes. See "*Breakpoints for Multiple Processes*" on page 203.

### totalview*stopForegroundColor: *color*

**Default: red**

Sets the color of **STOP** and **ASM** icons to *color*.

### totalview*stoppedFontForegroundColor: *color*

Default: **blue** or **yellow2**

Sets the color of "**T**" state to *color*.

### totalview*tmpFile1HeaderString: *string*

The header line used within the first temporary file used when TotalView does a bulk server launch operation. See "*%t1 and %t2*" on page 317.

### totalview*tmpFile1HostString: *string*

The host line used within the first temporary file used when TotalView does a bulk server launch operation. See "*%t1 and %t2*" on page 317.

### totalview*tmpFile1TrailerString: *string*

The trailer line used within the first temporary file used when TotalView does a bulk server launch operation. See "*%t1 and %t2*" on page 317.

### totalview*tmpFile2HeaderString: *string*

The header line used within a temporary file used when TotalView does a bulk server launch operation. See "*%t1 and %t2*" on page 317.

**totalview*tmpFile2HostString**: *string*

The host line used within the second temporary file used when TotalView does a bulk server launch operation. See "*%t1 and %t2*" on page 317.

**totalview*tmpFile2TrailerString**: *string*

The trailer line used within the second temporary file used when TotalView does a bulk server launch operation. See "*%t1 and %t2*" on page 317.

**totalview*useInterface**: *name*

Sets the interface name that the server uses when it makes a call back. For example, on an IBM PS2 machine, the following resource setting sets the callback to use the hardware option:

```
totalview*useInterface:css0
```

However, TotalView will let you use any legal **inet** interface name. (You can obtain a list of the interfaces if you use the **netstat -i** command.)

**totalview*useColor**: {**true** | **false**}

If **true** (default), enables TotalView use of color. If **false**, disables all use of color and display using monochrome black on white. This option overrides all other color-related options.

Override with: **–color** option (overrides **false**)
**–no_color** option (overrides **true**)

**totalview*userThreads**: {**true** | **false**}

If set to **true** (default), enables handling of user-level (M:N) thread packages on systems where two-level (kernel and user) thread scheduling is supported. If set to **false**, TotalView disables handling of user-level (M:N) thread packages. Disabling thread support may be useful in situations where you need to debug kernel-level threads, but in most cases, this option is of little use on systems where two-level thread scheduling is used.

Override with: **–user_threads** option (overrides **false**)
**–no_user_threads** option (overrides **true**)

**totalview*useTextColor**: {**true** | **false**}

If **true** (default), enables TotalView use of text color. If **false**, TotalView disables use of text color.

Override with: **–text_color** option (overrides **false**)
**–no_text_color** option (overrides **true**)

**totalview*useTitleColor**: {**true** | **false**}

If **true** (default), enables TotalView use of title color. If **false**, TotalView disables use of title color.

Override with: **–title_color** option (overrides **false**)
**–no_title_color** option (overrides **true**)

**totalview*useTransientFor**: {**on** | **off**}

If **off**, use "override redirect" windows, which doesn't let you use the window manager to perform operations, such as raise and lower, on dialog boxes. If you use an advanced window manager, you can use the **on** option (default) to specify that the debugger use "transient-for" type windows, which allow you to use the window manager to perform operations on dialog boxes. If you're using an X11R4 or more recent server and window manager, you should use the **on** option. If you're using Compaq's window manager, you should use the **off** option.

**totalview*verbosity**: {silent | error | warning | info}

**Default: info**

Sets the verbosity level of TotalView generated messages.

**totalview*visualizerLaunchString**: *command_string*

**Default: visualize**

Specifies the command string that TotalView uses to launch the visualizer when you first visualize something. This is a shell command line, so you can use the shell redirection command to output visualization data-sets to a file (for example, "**cat >** *your_file*").

**totalview*visualizerLaunchEnabled**: {**true** | **false**}

If **true** (default), TotalView automatically launches the visualizer when you first visualize something. If **false**, TotalView disables visualization.

**totalview*visualizerMaxRank**: *n*

**Default**: 2

Specifies the default value used in the "**Maximum permissible rank**" field of the Visualizer Launch Window dialog box. This field sets the maximum rank of the array that TotalView will export to the visualizer. TotalView's default visualizer cannot visualize arrays of rank greater than two, however if you are using another visualizer, or just dumping binary data, you can set the limit here.

**totalview*warnStepThrow**: {**true** | **false**}

If set to **true** (default), and your program throws an exception during a TotalView single-step operation, you will be asked if you wish to stop the single-step operation. The process will be left stopped at the C++ run time library's "throw" routine. If set to **false**, then TotalView will not catch C++ exception throws during single-step operations, which may cause the single-step operation to lose control of the process, and cause it to run away.

# Visualizer X Resources

The TotalView visualizer uses a large number of X resources that are set up in its application defaults file. The X resources documented are a subset of those found in the application defaults file as they are the only ones that may be customized to your preferences. Setting them in your own X resources file overrides the application defaults file.

The default values of the X resources are listed here shown either in a bold typeface in a list of alternative values, or separately if there can be a range of values. They are the settings in the applications defaults file as it is shipped. Your site administrator can edit this file to set the site defaults, therefore your site may have different defaults.

**Visualize*data*pick_message.background**: *color*

**Default**: **light yellow**

Sets the color of the pick popup window.

**Visualize*directory*auto_visualize.set**: {1 | 0}

Sets the initial state of the auto-visualize option in the Directory Window. If set (1), when a new data-set is added to the list, it will be visualized automatically using an appropriate method. If cleared (0), the new data-set will not be displayed automatically, and you will have to choose a visualization method for it.

**Visualize*directory.width**:

*width*
Visualize*directory.height

*height*

Sets the initial width and height of the Directory Window.

Default: *width*=**300**, *height*=**100**

**Visualize*graph.width**:

Default: width=**400**, height=**400**

*width*
Visualize*graph.height

*height*

Sets the initial width and height of the Graph Data Window.

**Visualize*graph*lines.set**: {1 | 0}

Sets the initial state of the lines option in the Graph Window. When set (1), graphs are drawn with lines connecting the data points.

**Visualize*graph*points.set**: {1 | 0}

Sets the initial state of the points option in the Graph Window. When set (1), graphs are drawn with markers on each data point.

**Visualize*surface.width**: *width*
**Visualize*surface.height**: *height*

Default: *width*=400, *height*=400

Sets the initial width and height of the Surface Data Window.

**Visualize*surface*mesh.set**: {1 | 0}

Sets the initial state of the mesh option in the Surface Window. When set (1), the axis grid is projected onto the surface.

**Visualize*surface*shade.set**: {1 | 0}

Sets the initial state of the shade option in the Surface Window. When set (1), the surface is shaded.

**Visualize*surface*contour.set**: {1 | 0}

Sets the initial state of the contour option in the Surface Window. When set (1), contours are displayed on the surface.

**Visualize*surface*zone.set**: {1 | 0}

Sets the initial state of the zone option in the Surface Window. When set (1), the surface is colored according to the value.

**Visualize*surface*auto_reduce.set**: {1 | 0}

Sets the initial state of the auto-reduce option in the Surface Window. When set (1), large data-sets are reduced by averaging to speed display.

**Visualize*surface*xrt3dZoneMethod**: {zonecontours | zonecells}

Specifies how the surface is colored. When set to **zonecontours**, the surface is colored according to its contours. When set to **zonecells**, each cell in the mesh is colored based on the average value in the cell.

**Visualize*surface*xrt3dViewNormalized**: {1 | 0}

When set (1), the view of the data-set (before zooming or translation) is maximized to fit the window. Interactive rotation when this resource is set will look "jerky" but will ensure no portion of the display is clipped. When this resource is cleared (0), dynamic rotation will be smooth, but parts of the display (for example, axes) may be clipped at some viewing angles.

**Visualize*surface*xrt3dXMeshFilter**: : *n*
**Visualize*surface*xrt3dYMeshFilter**: *n*

Default: 0

Specifies how to display the surface mesh. Every *n*th mesh line will be displayed, where *n* must be an integer greater than or equal to 0. When set to **0**, a value is calculated automatically.

Chapter 12

# TotalView Command Syntax

**This chapter summarizes the syntax of the totalview command. For the full syntax, use the man totalview command to view the online version.**

## Syntax

**Synopsis:** **totalview** [ *filename* [ *corefile* ]] [ *options* ]

**Description:** The TotalView debugger is a source-level debugger with a graphic interface (based on the X Window System) and features for debugging distributed programs, multiprocess programs, and multithreaded programs. You need a workstation or terminal running the X Window System to use TotalView. TotalView is available on a number of different platforms.

**Arguments:**

*filename* Specifies the pathname of the executable being debugged. This can be an absolute or relative pathname. The executable must be compiled with debugging symbols turned on, normally the **–g** compiler option. Any multiprocess programs that call **fork()**, **vfork()**, or **execve()** should be linked with the **dbfork** library.

*corefile* Specifies the name of a core file. Specify this argument in addition to *filename* when you want to examine a core file with TotalView:

**totalview** *filename corefile* [ *options* ]

**Using Options :** If you specify mutually exclusive options on the same command line (for example, **–dynamic** and **–no_dynamic**), the last option listed is used. Some of these options override TotalView X resources described in "*X Resources*" on page 275. If an option contains underscores (_), you can usually omit the underscores. For example, **–nodynamic** is the same as **–no_dynamic**; similarly **–arrowbgcolor** and **–arrow_bg_color** are the same.

**NOTE** The option, –X*resource*=*value*, allows you to set the X resource X*resource* to *value* from the command line. For example, to set totalview*stopAll to false, you could specify the command line option –stopAll=false. Note that the string "totalview*" is omitted from the command line. X resource values set from the command line override settings in your X resource file. For a complete list of X resources, see Chapter 11 "*X Resources*" on page 275.

# Options

**–a** *args*
Passes all subsequent arguments (specified by *args*) to the program specified by *filename*. This option must be the *last* one on the command line.

**–arrow_bg_color** *color*
Sets the background (outline) color of PC arrow to *color*.

**Default: black**

**–arrow_color** *color*
Sets the foreground (inner) color of PC arrow to *color*.

**Default: yellow2**

**–ask_on_dlopen**
(Default) TotalView will ask you about stopping processes that dynamically load a new shared library using the **dlopen** or **load** (AIX only) system calls. See "*Debugging Dynamically Loaded Libraries*" on page 338.

**–no_ask_on_dlopen**
TotalView will *not* ask you about stopping processes that dynamically load a new shared library using the **dlopen** or **load** (AIX only) system calls. See "*Debugging Dynamically Loaded Libraries*" on page 338.

**–background** *color* Sets the general background color to *color*.

**Default: white**

**–bg** *color* Same as **–background**.

**–barrier_color** *color*

Sets the color of the process barrier breakpoint icon.

**Default: blue**

**–barrier_font_color** *color*

Sets the color of the font used to show the **H** and **Hold** indicators for held processes.

**Default: blue**

**–barr_stop_all** (Default) Enables process barrier breakpoints to stop all related processes.

**–no_barr_stop_all**

The process barrier breakpoint does *not* stop all related processes.

**–break_color** *color* Sets the color of "**B**" state to *color*.

**Default: orange**

**–button_bg_color** *color*

Sets the button background color to *color*.

Default: background color

**–button_fg_color** *color*

Sets the button foreground color to *color*.

Default: foreground color

**–chase** (Default) Displays dialog boxes at the mouse pointer. To display dialog boxes centered in the upper third of the screen, use **–no_chase**.

**–no_chase** Displays dialog boxes centered in the upper third of the screen.

**–color** (Default) Enables TotalView use of color.

**–no_color** Disables all use color, and display using monochrome black on white. This option overrides all other color-related options.

**–nc** Same as **–no_color**.

**–compiler_vars** Alpha, HP, and SGI only. Show variables created by the Fortran compiler, as well as those in the user's program.

**–no_compiler_vars**
(Default) Do not show variables created by the Fortran compiler.

Some Fortran compilers (Compaq f90/f77, HP f90, SGI 7.2 compilers) output debug information which describes variables that the compiler itself has invented for purposes such as passing the length of character*(*) variables. By default, TotalView suppresses the display of these compiler generated variables.

However you can specify the **–compiler_vars** option or set the **totalview*compilerVars** X resource to true to cause such variables to be displayed. This could be useful if you are looking for a corruption of a run time descriptor or are writing a compiler.

**–dbfork** (Default) Catches the **fork()**, **vfork()**, and **execve()** system calls if your executable is linked with the **dbfork** library.

**–no_dbfork** Does not catch **fork()**, **vfork()**, and **execve()** system calls even if your executable is linked with the **dbfork** library.

**–debug_file** *consoleoutputfile*
Redirects TotalView console output to a file named *consoleoutputfile*.

Default: All TotalView console output is written to **stderr**.

**–demangler=***compiler*

Overrides the C++ demangler and mangler TotalView uses by default. The following table lists override options.

TABLE 39: Demangling Command Line Options

| Option | Meaning |
|---|---|
| **–demangler=cset** | IBM xlC C++ |
| **–demangler=dec** | Compaq Tru64 C++ |
| **–demangler=gnu** | GNU C++ |
| **–demangler=hpux** | HP C++ |
| **–demangler=irix** | SGI IRIX C++ |
| **–demangler=kai** | KAI KCC C++ 3.2 or greater |
| **–demangler=spro** | SunPro C++ 4.0 or greater |
| **–demangler=sun** | Sun CFRONT C++ |
| **–demangler=usoft** | Microsoft C++ |

**–display** *displayname*

Sets the name of the X Windows display to *displayname*. For example, **–display vinnie:0.0** will display TotalView on the machine named "vinnie."

Default: To the value of the **DISPLAY** environment variable.

**–dpvm** Compaq Tru64 UNIX only: Enables support for debugging the Compaq Tru64 UNIX implementation of Parallel Virtual Machine (PVM) applications.

**–no_dpvm** Compaq Tru64 UNIX only: (Default) Disables support for debugging the Compaq Tru64 UNIX implementation of PVM applications.

**–dump_core** Allows TotalView to dump a core file when it gets an internal error. Useful for debugging TotalView itself.

**–no_dumpcore** (Default) Does not allow TotalView to dump a core file when it gets an internal error.

**–dynamic** (Default) Loads symbols from shared libraries. This option is available only on platforms that support shared libraries.

| | |
|---|---|
| **–no_dynamic** | Does not load symbols from shared libraries when reading dynamically linked executables. Setting this option can cause the **dbfork** library to fail because TotalView might not find the **fork()**, **vfork()**, and **execve()** system calls. |
| **–error_color** *color* | Sets the color of "**E**", "**Z**", and "**?**" states to *color*.<br>**Default: red** |
| **–eval_color** *color* | Sets the color of the EVAL action point signs to *color*.<br>**Default: orange** |
| **–ext** *extension* | Specifies that files with the suffix *extension* are preprocessor input files. TotalView already has built-in extensions for C++ (**.C**, **.cpp**, **.cc**, **.cxx**), Fortran (**.F**), **lex** (**.l**, **.lex**), and **yacc** (**.y**) files. |
| **–font** *fontname* | Specifies the font to be used by TotalView.<br>**Default: fixed** |
| **–fn** *fontname* | Same as **–font**. |
| **–foreground** *color* | Sets the general foreground color (i.e., the text color) to *color*.<br>**Default: black** |
| **–fg** *color* | Same as **–foreground**. |
| **–global_types** | (Default) Specifies that TotalView can assume that type names are globally unique within a program and that all type definitions with the same name are identical. In C++, the standard asserts that this must be true for standard conforming code.<br>If this option is set, TotalView will attempt to replace an opaque type (**struct foo *p;**) declared in one module, with an identically named defined type in a different module.<br>If TotalView has read the symbols for the module containing the non-opaque type definition, then when displaying variables declared with the opaque type, TotalView will automatically display the variable using the non-opaque type definition. |

**–no_global_types** Specifies that TotalView *cannot* assume that type names are globally unique within a program. You should specify this option if your code has multiple different definitions of the same named type, since otherwise TotalView is likely to pick the wrong definition to substitute for an opaque type.

**–grab** Forces all keyboard input to go to an open dialog box. Use this option if your window manager uses "click-to-type" mode.

**–no_grab** (Default) Does not force keyboard input to an open dialog box.

**–grab_server** (Default) TotalView will grab the X server when posting menus.

**–no_grab_server**
TotalView will not grab the X server when posting menus. Useful for taking screen shots of TotalView's menus.

**–hpf** (Default) Enables debugging HPF code at the source level.

**–no_hpf** Disables debugging HPF source code at the source level.

**–hpf_node** Enables display of node on which HPF distributed array element resides in the Process Window.

**–no_hpf_node** (Default) Disables display of node on which HPF distributed array element resides in the Process Window.

**–ignore_control_c** Ignores Ctrl-C and prevents you from terminating the TotalView process from an **xterm** window, which is useful when your program catches the Ctrl-C signal (SIGINT).

**–icc** Same as **–ignore_control_c**.

**–no_ignore_control_c**
(Default) Catches Ctrl-C and terminates your TotalView debugging session. To override this, use **–ignore_control_c**.

**–nicc** Same as **–no_ignore_control_c**.

**–iv** Turns inverse video on.

**–no_iv** (Default) Turns inverse video off.

**–kcc_classes** (Default) Convert structure definitions output by the KCC compiler into classes that show base classes, and virtual base classes in the same way as other C++ compilers. See the description of the X resource "TOTALVIEW*KCCCLASSES" on page 284 for a description of the conversion performed by TotalView.

**–no_kcc_classes**
Do not convert structure definitions output by the KCC compiler into classes. Virtual bases will show up as pointers, rather than the data.

**–lb** (Default) Loads action points automatically from the *filename*.**TVD.breakpoints** file, providing the file exists.

**–nlb** Does not load action points automatically from an action points file.

**–mc** Turns on menu caching. Use this option if your X server supports menu caching. If menus appear blank the second and subsequent times you display them, your X server does not support menu caching.

**–nmc** (Default) Turns off menu caching.

**–menu_arrow_color** *color*
Sets the menu arrow color to *color*.

Default: **blue** or **green**

**–message_queue** (Default) Enable the display of MPI message queues when debugging an MPI program.

**–mqd** Same as **–message_queue**.

**–no_message_queue**
Disable the display of MPI message queues when debugging an MPI program. This might be useful if a store corruption is overwriting the message queues and causing TotalView to become confused.

**–no_mqd** Same as **–no_message_queue**.

**–mouse_bg_color** *color*
Sets the mouse cursor background (mask) color to *color*.

Default: **white** or **black**

**–mouse_fg_color** *color*
Sets the mouse cursor foreground (inner) color to *color*.

**Default: red**

**–mult_color** *color* Sets the color of MULT action point sign to *color*.

**Default: purple**

**–parallel** (Default) Enable handling of parallel program runtime libraries such as MPI, PE and HPF.

**–no_parallel** Disable handling of parallel program runtime libraries such as MPI, PE and HPF. This is useful for debugging parallel programs as if they were single process programs.

**–patch_area_base** *address*
Allocates the patch space dynamically at the given *address*. See "*Allocating Patch Space for Compiled Expressions*" on page 220.

**–patch_area_length** *length*
Set the length of the dynamically allocated patch space to the specified *length*. See "*Allocating Patch Space for Compiled Expressions*" on page 220.

**–pop_at_breakpoint**
Sets the **Open (or raise) process window at breakpoint** checkbox to be selected by default. See "*Handling Signals*" on page 41.

**–no_pop_at_breakpoint**
(Default) Sets the **Open (or raise) process window at breakpoint** checkbox to be deselected by default.

**–pop_on_error** (Default) Sets the **Open (or raise) process window on error** checkbox to be selected by default. See "*Handling Signals*" on page 41.

**–no_pop_on_error**
Sets the **Open (or raise) process window on error** checkbox to be deselected by default.

**–pr** Use pull-right menus.

**–npr** (Default) Use walking menus instead of pull-right menus.

**–pvm** Enables support for debugging the ORNL implementation of Parallel Virtual Machine (PVM) applications.

**–no_pvm** (Default) Disables support for debugging the ORNL implementation of PVM applications.

**–remote** *hostname*[:*portnumber*]

Debugs an executable that is not running on the same machine as TotalView. For *hostname*, you can specify a TCP/IP hostname, such as **vinnie**, or a TCP/IP address, such as **128.89.0.16**. Optionally, you can specify a TCP/IP port number for *portnumber*, such as **:4174**. When you specify a port number, you disable the autolaunch feature. For more information on the autolaunch feature, see "*Single Process Server Launch Command*" on page 59.

**–r** *hostname*[:*portnumber*]

Same as **–remote**.

**–running_color** *color*

Sets the color of "**R**", "**S**", "**M**", and "**I**" states to *color*.

**Default: green**

**-s** *pathname* Specifies the pathname of a start-up file that will be loaded and executed. This pathname can either be an absolute or relative name. You can find information on the contents of this start-up file in the CLI *User's Guide*.

**–sb** Saves action points automatically to an action points file when you exit TotalView. The file is named *filename*.**TVD.breakpoints**.

**–nsb** (Default) Does not save action points automatically to an action points file when you exit.

**–serial** *device*[:*options*]

Debugs an executable that is not running on the same machine as TotalView. For *device*, specify the device name of a serial line, such as **/dev/com1**. Currently, the only *option* you are allowed to specify is the baud rate, which defaults to **38400**. For more information on debugging over a serial line, see "*Debugging Over a Serial Line*" on page 65.

**–signal_handling_mode "***action_list***"**
Modifies the way in which TotalView handles signals. You must enclose the *action_list* string in quotation marks to protect it from the shell. Refer to "TOTALVIEW*SIGNALHANDLINGMODE" on page 290 for a description of the *action_list* argument.

**–shm "***action_list***"** Same as **–signal_handling_mode**.

**–stop_all** (Default) Sets the **Stop All Related Processes when Breakpoint Hit** checkbox to be selected by default. To override this option use **–no_stop_all**. See "*Breakpoints for Multiple Processes*" on page 203.

**–no_stop_all** Sets the **Stop All Related Processes when Breakpoint Hit** checkbox to be deselected by default.

**–stop_color** *color* Sets the color of STOP and ASM action point signs to *color*.

**Default: red**

**–stopped_color** *color*
Sets the color of "**T**" state to *color*.

Default: **blue** or **yellow2**

**–text_color** (Default) Turns text color use on.

**–no_text_color** Turns text color use off.

**–title_color** (Default) Turns title color use on.

**–tc** Same as **–title_color**.

**–no_title_color** Turns title color use off.

**–no_tc** Same as **–no_title_color**.

**–user_threads** (Default) Enable handling of user-level (M:N) thread packages on systems where two-level (kernel and user) thread scheduling is supported.

**–no_user_threads**
Disable handling of user-level (M:N) thread packages. This option may be useful in situations where you need to debug kernel-level threads, but in most cases, this

option is of little use on systems where two-level thread scheduling is used.

**–verbosity** *level* — Sets the verbosity level of TotalView generated messages to *level*, which may be one of **silent**, **error**, **warning**, or **info**.

Default: **info**

Chapter 13

# TotalView Debugger Server Command Syntax

This chapter summarizes the syntax of the TotalView Debugger Server command, tvdsvr, which is used for remote debugging. For more information on remote debugging, refer to "*Starting the Debugger Server for Remote Debugging*" on page 55.

## The tvdsvr Command and its Options

**Synopsis :** tvdsvr {**–server** | **–callback** *hostname*:*port* | **–serial** *device*} [*other options*]

**Description :** The **tvdsvr** debugger server allows TotalView to control and debug a program on a remote machine. To accomplish this, the **tvdsvr** program must run on the remote machine, and it must have access to the executables to be debugged. These executables must have the same absolute pathname as the executable that TotalView is debugging, or the PATH environment variable for **tvdsvr** must include the directories containing the executables.

You must specify either the **–server**, **–callback**, or **–serial** option with the **tvdsvr** command. By default, the TotalView debugger automatically launches **tvdsvr** (known as the auto-launch feature) with the **–callback** option, and the server establishes a connection with TotalView.

If you prefer not to use the auto-launch feature, you can start **tvdsvr** manually and specify the **–server** option. Be sure to note the password that **tvdsvr** prints out with the message:

**pw** = *hexnumhigh*:*hexnumlow*

TotalView will prompt you for *hexnumhigh*:*hexnumlow* later. By default, **tvdsvr** automatically generates a password that is used when establishing connections. If desired, you can use the **–set_pw** option to set a specific password.

To connect to the **tvdsvr** from TotalView, you use the New Program Window and must specify the hostname and TCP/IP port number, *hostname*:*portnumber* on which **tvdsvr** is running. Then, TotalView prompts you for the password for **tvdsvr**.

**Options:** The following options determine the port number and password necessary for TotalView to connect with **tvdsvr**.

**–callback** *hostname***:***port*

(Auto-launch feature only) Immediately establishes a connection with a TotalView process running on *hostname* and listening on *port*, where *hostname* is either a hostname or TCP/IP address. If **tvdsvr** cannot connect with TotalView, it exits.

If you use a **–port, –search_port**, or **–server** options with this option, **tvdsvr** ignores them.

**–callback_host** *hostname*

Names the host upon which the callback is made. *hostname* indicates the machine upon which TotalView is running. This option is most often used with a bulk launch.

**–callback_ports** *port-list*

Names the ports on the host machines that are used for callbacks. The *port-list* argument contains a comma-separated list of the host names and TCP/IP port numbers (*hostname*:*port*,*hostname*:*port*...) on which TotalView is listening for connections from **tvdsvr**. This option is most often used with a bulk launch.

**–debug_file** *consoleoutputfile*
Redirects TotalView Debugger Server console output to a file named *consoleoutputfile*.

Default: All console output is written to **stderr**.

**–dpvm**
Uses the Compaq Tru64 UNIX implementation of the Parallel Virtual Machine (DPVM) library process as its input channel and registers itself as the DPVM tasker.

NOTE This option is not intended for users launching **tvdsvr** manually. When you enable DPVM support within TotalView, TotalView automatically uses this option when it launches tvdsvr.

**–port** *number*
Sets the TCP/IP port number on which **tvdsvr** should communicate with **totalview**. If this TCP/IP port number is busy, **tvdsvr** does not select an alternate port number (that is, it communicates with nothing) unless you also specify **–search_port**.

Default: 4142

**–pvm**
Uses the ORNL implementation of the Parallel Virtual Machine (PVM) library process as its input channel and registers itself as the ORNL PVM tasker.

NOTE This option is not intended for users launching **tvdsvr** manually. When you enable PVM support within TotalView, TotalView automatically uses this option when it launches tvdsvr.

**–search_port**
Searches for an available TCP/IP port number, beginning with the default port (4142) or the port set with the **–port** option and continuing until one is found. When the port number is set, **tvdsvr** displays the chosen port number with the following message:

**port** = *number*

Be sure that you remember this port number as you will need it when you are connecting to this server from TotalView.

**–serial** *device*[:*options*]

Waits for a serial line connection from TotalView. For *device*, specify the device name of a serial line, such as **/dev/com1**. The only *option* you can specify is the baud rate, which defaults to **38400**. For more information on debugging over a serial line, see "*Debugging Over a Serial Line*" on page 65.

**–server**

Listens for and accepts network connections on port 4142 (default).

Using **–server** can be a security problem. Consequently, you must explicitly enable this feature by placing an empty file named **tvdsvr.conf** in your **/etc** directory. This file must be owned by user ID 0 (root). When the **tvdsvr** encounters this option, it checks if this file exists. This file's contents are ignored.

You can use a different port by specifying either **–port** or **–search_port**. To stop **tvdsvr** from listening and accepting network connections, you must terminate it by pressing Ctrl-C in the terminal window from which it was started or by using the **kill** command.

**–set_pw** *hexnumhigh*:*hexnumlow*

Sets the password to the 64-bit number specified by the two 32-bit numbers *hexnumhigh* and *hexnumlow*. When a connection is established between **tvdsvr** and TotalView, the 64-bit password passed by TotalView must match the password set with this option. When the password is set, **tvdsvr** displays the selected number in the following message:

**pw =** *hexnumhigh*:*hexnumlow*

We recommend using this option to avoid connections by other users.

NOTE **If necessary, you can disable password checking by specifying the** *–set_pw* 0:0 **option with the** *tvdsvr* **command. Disabling password checking is dangerous; it allows anyone to connect to your server and start programs, including shell commands, using your UID. Therefore, we do not recommend disabling password checking.**

**–set_pws** *password-list*

Sets 64-bit passwords. TotalView must supply these passwords when **tvdsvr** establishes the connection with it. The argument to this command is a comma-separated list of passwords that TotalView automatically generates. This option is most often used with a bulk launch.

**–verbosity** *level*

Sets the verbosity level of TotalView Debugger Server generated messages to *level*, which may be one of **silent**, **error**, **warning**, or **info**.

Default: **info**

**–working_directory** *directory*

Makes *directory* the directory to which TotalView will be connected.

Note that the command assumes that the host machine and the target machine mount identical file-systems. That is, the pathname of the directory to which TotalView is connected must be identical on both the host and target machines.

After performing this operation, the TotalView Debugger Server is started.

# Replacement Characters

When placing a **tvdsvr** command within a **Server Launch** or **Bulk Launch** Window, you will need to use special replacement characters. When your program needs to launch a remote process, TotalView replaces these characters within the command with what it represents. Here are the replacement characters:

**%C**

Is replaced by the name of the server launch command being used. On most platforms, this is **rsh**. On HP, this command is **remsh**. If the **TVDSVRLAUNCHCMD** environment variable exists, TotalView will use its value instead of its platform-specific value.

| | |
|---|---|
| %D | Is replaced by the absolute pathname of the directory to which TotalView will be connected. |
| %H | Expands to the hostname of the machine upon which TotalView is running. (These replacement characters are most often used in bulk server launch commands. However, it can be used in a regular server launch and within a **tvdsvr** command contained within a temporary file.) |
| %L | If TotalView is launching one process, this is replaced by the host name and TCP/IP port number (*hostname*:*port*) on which TotalView is listening for connections from **tvdsvr**.<br><br>If a bulk launch is being performed, TotalView replaces this with a comma-separated list of the host names and TCP/IP port numbers (*hostname*:*port*,*hostname*:*port*...) on which TotalView is listening for connections from **tvdsvr**. |
| %N | Is replaced by the number of servers that will be launched. This is only used in a bulk server launch command. |
| %P | If TotalView is launching one process, this is replaced by the password that TotalView automatically generated.<br><br>If a bulk launch is being performed, TotalView replaces this with a command separated list of 64-bit passwords |
| %R | Is replaced by the host name of the remote machine that was specified in the **New Program Window** command. |
| %S | If TotalView is launching one process, this is replaced by the port number on the machine upon which the debugger is running.<br><br>If a bulk server launch is being performed, TotalView replaces this with a comma-separated list of port numbers. |

**%t1** and **%t2** Is replaced by files that TotalView creates containing information it generates. This is only available in a bulk launch.

These temporary files have the following structure:

(1) An optional header line containing initialization commands required by your system.

(2) One line for each host being connected to, containing host-specific information.

(3) An optional trailer line containing information needed by your system to terminate the temporary file.

The bulk server launch dialog box allows you to define templates for the actions performed by temporary files. These files will use these replacement characters. You can only use the **%N**, **%t1**, and **%t2** replacement characters within header and trailer lines of temporary files. The **%L**, **%P**, and **%S** can be used in header or trailer lines or within a host line defining the command that initiates a single process server launch.

The templates for temporary files can also be set using X Resources. These resources, described in Chapter 11, all begin with "totalview*tmpFile."

**%V** Is replaced by the current TotalView verbosity setting.

Appendix A

# Compilers and Environments

This appendix describes the compilers and parallel runtime environments that can be used with this release of TotalView. You must refer to the TotalView release notes included in the TotalView distribution for information on the specific compiler and runtime environment supported by TotalView.

For information on supported operating systems, please refer to Appendix B "*Operating Systems*" on page 329.

This appendix includes:

- Compiling with exception data on Compaq Tru64 UNIX
- Linking with the dbfork library

## Compiling with Debugging Symbols

You need to compile programs with the **–g** option and possibly other compiler options so that debugging symbols are included. This section shows the specific compiler commands to use for each compiler that TotalView supports.

**NOTE** Please refer to the release notes in your TotalView distribution for the latest information about supported versions of the compilers and parallel runtime environments listed here.

## AIX on RS/6000 Systems

Table 40 lists the procedures to compile programs on IBM RS/6000 systems running AIX.

TABLE 40: Compiling with Debugging Symbols on AIX

| Compiler | Compiler Command Line |
|---|---|
| GCC EGCS C | **gcc –g –c** *program*.**c** |
| GCC EGCS C++ | **g++ –g –c** *program*.**cxx** |
| IBM xlc C | **xlc –g –c** *program*.**c** |
| IBM xlC C++ | **xlC –g –c** *program*.**cxx** |
| IBM xlf Fortran 77 | **xlf –g –c** *program*.**f** |
| IBM xlf90 Fortran 90 | **xlf90 –g –c** *program*.**f90** |
| KAI C | **KCC +K0 –qnofullpath –c** *program*.**c** |
| KAI C++ | **KCC +K0 –qnofullpath –c** *program*.**cxx** |
| KAI Guide C (OpenMP) | **guidec -g +K0** *program*.**c** |
| KAI Guide C++ (OpenMP) | **guidec -g +K0** *program*.**cxx** |
| KAI Guide F77 (OpenMP) | **guidef77 -g -WG,-comp=i** *program*.**f** |
| Portland Group HPF | **pghpf –g –Mtv –c** *program*.**hpf** |

When compiling with KCC, you must specify the **–qnofullpath** option; KCC is a preprocessor that passes its output to the IBM xlc C compiler. It will discard **#line** directives necessary for source level debugging if **–qfullpath** is specified. We also recommend that you use the **+K0** option and not the **–g** option.

When compiling with **guidef77**, the **-WG,-comp=i** option may not be required on all versions because **-g** can imply these options.

When compiling Fortran programs using the C preprocessor, pass the **–d** option to the compiler driver. For example: **xlf –d –g –c program.F**

When compiling with any of the IBM xl compilers, if your program will be moved from its creation directory, or you do not want to set the search directory path during debugging, use the **–qfullpath** compiler option. For example:

```
xlf –qfullpath –g –c program.f
```

## Compaq Tru64 UNIX

Table 41 lists the procedures to compile programs on Compaq Tru64 UNIX.

TABLE 41: **Compiling with Debugging Symbols on Compaq Tru64 UNIX**

| Compiler | Compiler Command Line |
|---|---|
| Compaq Tru64 UNIX C | **cc –g –c** *program*.**c** |
| Compaq Tru64 UNIX C++ | **cxx –g –c** *program*.**cxx** |
| Compaq Tru64 UNIX Fortran 77 | **f77 –g –c** *program*.**f** |
| Compaq Tru64 UNIX Fortran 90 | **f90 –g –c** *program*.**f90** |
| GCC EGCS C | **gcc –g –c** *program*.**c** |
| GCC EGCS C++ | **g++ –g –c** *program*.**cxx** |
| KAI C | **KCC +K0 –c** *program*.**c** |
| KAI C++ | **KCC +K0 –c** *program*.**cxx** |
| KAI Guide C (OpenMP) | **guidec -g +K0** *program*.**c** |
| KAI Guide C++ (OpenMP) | **guidec -g +K0** *program*.**cxx** |
| KAI Guide F77 (OpenMP) | **guidef77 -g -WG,-comp=i** *program*.**f** |

When compiling with KCC for debugging, we recommend that you use the **+K0** option and not the **–g** option. Also, the **-WG,-comp=i** option to the **guidef77** command may not be required on all versions because **-g** can imply these options.

## HP-UX

Table 42 lists the procedures to compile programs on HP-UX.

TABLE 42: **Compiling with Debugging Symbols on HP-UX**

| Compiler | Compiler Command Line |
|---|---|
| HP ANSI C | **cc –g –c** *program*.**c** |
| HP C++ | **aCC –g –c** *program*.**cxx** |
| HP Fortran 90 | **f90 –g –c** *program*.**f90** |
| KAI C | **KCC +K0 –c** *program*.**c** |
| KAI C++ | **KCC +K0 –c** *program*.**cxx** |
| KAI Guide C (OpenMP) | **guidec -g +K0** *program*.**c** |

TABLE 42: Compiling with Debugging Symbols on HP-UX (cont.)

| Compiler | Compiler Command Line |
|---|---|
| KAI Guide C++ (OpenMP) | **guidec -g +K0** *program*.**cxx** |
| KAI Guide F77 (OpenMP) | **guidef77 -g -WG,-comp=i** *program*.**f** |

When compiling with KCC for debugging, we recommend that you use the **+K0** option and not the **–g** option. Also, the **-WG,-comp=i** option to the **guidef77** command may not be required on all versions because **-g** can imply these options.

## IRIX on SGI MIPS Systems

Table 43 lists the procedures to compile programs on SGI MIPS systems running IRIX.

TABLE 43: Compiling with Debugging Symbols on IRIX-MIPS

| Compiler | Compiler Command Line |
|---|---|
| GCC EGCS C | **gcc –g –c** *program*.**c** |
| GCC EGCS C++ | **gcc –g –c** *program*.**cxx** |
| KAI C | **KCC +K0 –c** *program*.**c** |
| KAI C++ | **KCC +K0 –c** *program*.**cxx** |
| KAI Guide C (OpenMP) | **guidec -g +K0** *program*.**c** |
| KAI Guide C++ (OpenMP) | **guidec -g +K0** *program*.**cxx** |
| KAI Guide F77 (OpenMP) | **guidef77 -g -WG,-comp=i** *program*.**f** |
| Portland Group HPF | **pghpf –g –64 –Mtv –c** *program*.**hpf** |
| SGI MIPSpro 90 | **f90 –n32 –g –c** *program*.**f90**<br>**f90 –64 –g –c** *program*.**f90** |
| SGI MIPSpro C | **cc –n32 –g –c** *program*.**c**<br>**cc –64 –g –c** *program*.**c** |
| SGI MIPSpro C++ | **CC –n32 –g –c** *program*.**cxx**<br>**CC –64 –g –c** *program*.**cxx** |
| SGI MIPSpro77 | **f77 –n32 –g –c** *program*.**f**<br>**f77 –64 –g –c** *program*.**f** |

Compiling with **–n32** or **–64** is supported. TotalView does not support compiling with **–32**, which is the default for some compilers. You must specify either **–n32** or **–64**.

When compiling with KCC for debugging, we recommend that you use the **+K0** option and not the **–g** option. Also, the **-WG,-comp=i** option to the **guidef77** command may not be required on all versions because **-g** can imply these options.

You must compiler your programs with the pghpf **–64** compiler option; on SGI IRIX, TotalView can debug 64-bit executables only.

## SunOS 5 on SPARC

Table 44 lists the procedures to compile programs on SunOS 5 SPARC.

TABLE 44: **Compiling with Debugging Symbols on SunOS 5**

| Compiler | Compiler Command Line |
|---|---|
| Apogee C | **apcc –g –c** *program*.**c** |
| Apogee C++ | **apcc –g –c** *program*.**cxx** |
| GCC EGCS C | **gcc –g –c** *program*.**c** |
| GCC EGCS C++ | **g++ –g –c** *program*.**cxx** |
| KAI C | **KCC +K0 –c** *program*.**c** |
| KAI C++ | **KCC +K0 –c** *program*.**cxx** |
| KAI Guide C (OpenMP) | **guidec -g +K0** *program*.**c** |
| KAI Guide C++ (OpenMP) | **guidec -g +K0** *program*.**cxx** |
| KAI Guide F77 (OpenMP) | **guidef77 -g -WG,-comp=i** *program*.**f** |
| Portland Group HPF | **pghpf –g –Mtv –c** *program*.**hpf** |
| SunPro/WorkShop C | **cc –g –c** *program*.**c** |
| SunPro/WorkShop C++ | **CC –g –c** *program*.**cxx** |
| SunPro/WorkShop Fortran 77 | **f77 –g –c** *program*.**f** |
| WorkShop Fortran 90 | **f90 –g –c** *program*.**f90** |

When compiling with KCC for debugging, we recommend that you use the **+K0** option and not the **–g** option. Also, the **-WG,-comp=i** option to the

**guidef77** command may not be required on all versions because **-g** can imply these options.

# Using Exception Data on Compaq Tru64 UNIX

If you receive the following error message when you load an executable into TotalView, you may need to compile your program so that exception data is included:

> "Cannot find exception information. Stack backtraces may not be correct."

To provide a complete stack backtrace in all situations, TotalView needs the exception data to be included in the compiled executable. To compile with exception data, you need to use the following options:

**cc –Wl,–u,_fpdata_size** *program.c*

where:

| | |
|---|---|
| **–Wl** | Passes the arguments that follow to another compilation phase (**–W**), which in this case is the linker (**l**). Each argument is separated by a comma (,). |
| **–u** | Causes the linker to mark the next argument (**_fpdata_size**) as undefined. |
| **_fpdata_size** | Marks the **_fpdata_size** variable as undefined, which forces the exception data into the executable. |

Compiling with exception data increases the size of your executable slightly. If you choose not to compile with exception data, TotalView can provide correct stack backtraces in most situations, but not in all situations.

# Linking with the dbfork Library

If your program uses the **fork()** and **execve()** system calls, and you want to debug the child processes, you need to link programs with the **dbfork** library.

## AIX on RS/6000 Systems

Add either the **-dbfork** or **-ldbfork_64** argument to the command that you use to link your programs. If you are compiling 32-bit code, use the following arguments:

- **/usr/totalview/lib/libdbfork.a –bkeepfile:/usr/totalview/lib/libdbfork.a**
- **–L/usr/totalview/lib –ldbfork –bkeepfile:/usr/totalview/lib/libdbfork.a**

For example:

```
cc –o program program.c \
        –L/usr/totalview/lib –ldbfork \
        –bkeepfile:/usr/totalview/lib/libdbfork.a
```

If you are compiling 64-bit code, use the following arguments:

- **/usr/totalview/lib/libdbfork_64.a \
  –bkeepfile:/usr/totalview/lib/libdbfork.a**
- **–L/usr/totalview/lib –ldbfork_64 \
  –bkeepfile:/usr/totalview/lib/libdbfork.a**

For example:

```
cc –o program program.c \
        –L/usr/totalview/lib –ldbfork \
        –bkeepfile:/usr/totalview/lib/libdbfork.a
```

When you use **gcc** or **g++**, use the **–Wl,–bkeepfile** option instead of using **–bkeepfile**, which will pass the same option to the binder. For example:

```
gcc –o program program.c –L/usr/totalview/lib –ldbfork \
        –Wl,–bkeepfile:/usr/totalview/lib/libdbfork.a
```

## Linking C++ Programs with dbfork

The binder option **–bkeepfile** currently cannot be used with the IBM xlC C++ compiler. The compiler passes all binder options to an additional pass called **munch**, which cannot handle the **–bkeepfile** option.

To work around this problem, we have provided the C++ header file **libdbfork.h**. You must include this file somewhere in your C++ program, in order to force the components of the **dbfork** library to be kept in your executable. The file **libdbfork.h** is included only with the RS/6000 version of TotalView. This means that if you are creating a program that will run on more than one platform, you should place the **include** within an **#ifdef** statement. For example:

```
#ifdef _AIX
#include "/usr/totalview/lib/libdbfork.h"
#endif
int main (int argc, char *argv[])
{
}
```

In this case, you would not use the **-bkkepfile** option and would instead link your program using one of the following options:

- **/usr/totalview/lib/libdbfork.a**
- **–L/usr/totalview/lib –ldbfork**

## Compaq Tru64 UNIX

Add one of the following arguments to the command that you use to link your programs:

- **/opt/totalview/lib/libdbfork.a**
- **–L/opt/totalview/lib –ldbfork**

For example:

```
cc –o program program.c –L/opt/totalview/lib –ldbfork
```

As an alternative, you can set the LD_LIBRARY_PATH environment variable and omit the **–L** option on the command line:

```
setenv LD_LIBRARY_PATH /opt/totalview/lib
```

## HP-UX

Add either the **-ldbfork** or **-ldbfork_64** argument to the command that you use to link your programs. If you are compiling 32-bit code, use one of the following arguments:

- **/opt/totalview/lib/libdbfork.a**
- **–L/opt/totalview/lib –ldbfork**

For example:

```
cc –n32 –o program program.c \
            –L/opt/totalview/lib –ldbfork
```

If you are compiling 64-bit code, use the following arguments:

- **/opt/totalview/lib/libdbfork_64.a**
- **–L/opt/totalview/lib –ldbfork_64**

For example:

```
cc –64 –o program program.c
    –L/opt/totalview/lib –ldbfork_64
```

As an alternative, you can set the **LD_LIBRARY_PATH** environment variable and omit the **–L** command line option. For example:

```
setenv LD_LIBRARY_PATH /opt/totalview/lib
```

## SunOS 5 SPARC

Add one of the following arguments to the command that you use to link your programs:

- **/opt/totalview/lib/libdbfork.a**
- **–L/opt/totalview/lib –ldbfork**

For example:

```
cc –o program program.c –L/opt/totalview/lib \
            –ldbfork
```

As an alternative, you can set the **LD_LIBRARY_PATH** environment variable and omit the **–L** option on the command line:

```
setenv LD_LIBRARY_PATH /opt/totalview/lib
```

## IRIX6-MIPS

Add one of the following arguments to the command that you use to link your programs.

If you are compiling your code with **–n32**, use the following arguments:

- **/opt/totalview/lib/libdbfork_n32.a**
- **–L/opt/totalview/lib –ldbfork_n32**

For example:

```
cc –n32 –o program program.c \
                –L/opt/totalview/lib –ldbfork_n32
```

If you are compiling your code with **–64**, use the following arguments:

- **/opt/totalview/lib/libdbfork.a_n64.a**
- **–L/opt/totalview/lib –ldbfork_n64**

For example:

```
cc –64 –o program program.c
     –L/opt/totalview/lib –ldbfork_n64
```

As an alternative, you can set the **LD_LIBRARY_PATH** environment variable and omit the **–L** option on the command line:

```
setenv LD_LIBRARY_PATH /opt/totalview/lib
```

Appendix B

# Operating Systems

This appendix describes the operating system features that can be used with TotalView. This appendix includes the following topics:

- Supported versions
- Mounting the /proc file system (Compaq Tru64 UNIX, IRIX, and SunOS 5 only)
- Swap space
- Shared libraries
- Remapping keys (Sun keyboards only)
- Capabilities and characteristics
- Expression system support

## Supported Operating Systems

For a complete list of hardware and software requirements including required OS patches and restrictions, see the TotalView release notes in your software distribution. This version of TotalView supports the following operating system versions:

- Compaq Alpha workstations running Compaq Tru64 UNIX versions V4.0B, V4.0C, V4.0D, V4.0E, V4.0F, and V5.0. All versions *require patches* See "Compaq UNIX Patch Procedures" in the TotalView Release Notes for instructions.
- HP PA-RISC 1.1 or 2.0 systems running HP-UX Version 11.0
- IBM RS/6000 and SP systems running AIX versions 4.2, 4.3, or 4.3.1
- Sun Sparc SunOS 5 (Solaris 2.x) systems running SunOS versions 5.5, 5.5.1, or 5.6. (Solaris 2.5, 2.5.1, or 2.6)

- SGI IRIX 6.2, 6.3, 6.4, or 6.5 on any MIPS R4000, R4400, R4600, R5000, R8000, or R10000 processor-based systems
- QSW CS-2 based on Sparc Solaris 2.5.1 or 2.6

**NOTE** TotalView on QSW CS-2 is nearly identical to TotalView on Sun Solaris 2.*x* systems.

# Mounting the /proc File System

## Compaq Tru64 UNIX, SunOS 5, and IRIX

To debug programs on Compaq Tru64 UNIX, SunOS 5, and IRIX with TotalView, you need to mount the **/proc** file system.

If you receive one of the following errors from TotalView, the **/proc** file system might not be mounted:

- `job_t::launch, creating process: process not found`
- `Error launching process while trying to read dynamic symbols`
- `Creating Process... Process not found`  
  `Clearing Thrown Flag`  
  `Operation Attempted on an unbound d_process object`

To determine whether the **/proc** file system is mounted, enter the appropriate command from the following table.

Table 45: Commands for Determining Whether */proc* is Mounted

| Operating System | Command |
|---|---|
| Compaq Tru64 UNIX | `%` **/sbin/mount –t procfs**<br>`/proc on /proc type procfs (rw)` |
| SunOS 5 | `%` **/sbin/mount \| grep /proc**<br>`/proc on /proc read/write/setuid on ...` |
| IRIX | `%` **/sbin/mount \| grep /proc**<br>`/proc on /proc type proc (rw)` |

If you receive the message shown from the mount command, the **/proc** file system is mounted.

### Compaq Tru64 UNIX and SunOS 5

To make sure that the **/proc** file system is mounted each time your system boots, add the appropriate line from the following table to the appropriate file.

Table 46: Commands for Automatically Mounting /proc File System

| Operating System | Name of File | Line to add |
|---|---|---|
| Compaq Tru64 UNIX | /etc/fstab | /proc /proc procfs rw 0 0 |
| SunOS 5 | /etc/vfstab | /proc - /proc proc - no - |

Then, to mount the **/proc** file system, enter the following command:

```
/sbin/mount /proc
```

### IRIX

To make sure that the **/proc** file system is mounted each time your system boots, make sure that **/etc/rc2** issues the **/etc/mntproc** command. Then, to mount the /proc file system, enter the following command:

```
/etc/mntproc
```

## Swap Space

Debugging large programs can exhaust the swap space on your machine. If you run out of swap space, TotalView exits with a fatal error, such as:

- Fatal Error: Out of space trying to allocate

  This error indicates that either TotalView failed to allocate dynamic memory. It can occur anytime during a TotalView session or that the data size limit in the C shell is too small. You can use the C shell's **limit** command to increase the data size limit. For example:

  ```
  limit datasize unlimited
  ```
- job_t::launch, creating process: Operation failed

This error indicates that the **fork()** or **execve()** system call failed while TotalView was creating a process to debug. It can happen when TotalView tries to create a process.

## Compaq Tru64 UNIX

To find out how much swap space has been allocated and is currently being used, use the **swapon** command on Compaq Tru64 UNIX:

```
% /sbin/swapon –s
Total swap allocation:
 Allocated space: 85170 pages (665MB)
 Reserved space: 14216 pages ( 16%)
 Available space: 70954 pages ( 83%)

Swap partition /dev/rz3b:
 Allocated space: 16384 pages (128MB)
 In-use space: 2610 pages ( 15%)
 Free space: 13774 pages ( 84%)

Swap partition /dev/rz3h:
 Allocated space: 52402 pages (409MB)
 In-use space: 2575 pages ( 4%)
 Free space: 49827 pages ( 95%)

Swap partition /dev/rz1b:
 Allocated space: 16384 pages (128MB)
 In-use space: 2592 pages ( 15%)
 Free space: 13792 pages ( 84%)
```

In this example, 665MB of swap has been allocated, and 106MB of it is currently in use.

To find out how much swap space is in use while you are running TotalView:

```
/bin/ps –o LFMT
```

For example, in this case the value in the VSZ column is 4.45MB:

```
  UID  PID PPID CP PRI NI    VSZ  RSS  ...
12270 5340 5293  0  41  0  4.45M 1.27  ...
```

To add swap space, use the **/sbin/swapon(8)** command. You must be **root** to use this command. For more information, refer to the on-line manual page for this command.

## AIX

To find out how much swap space has been allocated and is currently being used, use the **pstat** command:

```
% /usr/sbin/pstat –s

PAGE SPACE:

      USED PAGES   FREE PAGES
        7555         115325
```

In this example, 122880 (7555 + 115325) pages of swap have been allocated. 7555 pages are currently in use and 115325 pages are free.

To find out how much swap space is in use while you are running TotalView:

1 Start TotalView with a large executable:

   **totalview** *executable*

2 Press Ctrl-Z to suspend TotalView.
3 Use the following command to see how much swap space TotalView is using:

   ```
   ps u
   ```

   For example, in this case the value in the SZ column is 5476KB:

   ```
   USER    PID %CPU %MEM    SZ  RSS    TTY  ...
   smith 15080  0.0 6.0 5476 5476  pts/1 ...
   ```

To add swap space, use the AIX system management tool, **smit**. Use the following path through the **smit** menus:

```
System Storage Management → Logical Volume Manager →
      Paging Space
```

## HP HP-UX

The **swapinfo** command on an HP-UX system lets you find out how much swap space is allocated and is being used. For example:

```
# /usr/sbin/swapinfo
             Kb      Kb      Kb   PCT  START/      Kb
TYPE      AVAIL    USED    FREE  USED   LIMIT RESERVE  PRI  NAME
dev     1048576       0 1048576    0%       0       -    1  /dev/vg00/lvol2
reserve       -  389240 -389240
memory  1178960  966564  212396   82%
```

To find out how much swap space is being used while Totalview is running, type:

```
/usr/bin/ps -lf
```

Here is an example of what you might see:

```
  F S    UID   PID   PPID  C PRI  NI     ADDR    SZ ...
 21 T    rtf  4414  13709  0 154  20  ce8d800 2764 ...
```

The **SZ** field shows the pages occupied by a program.

To add swap space, use the**/usr/sbin/swapon**(1M) command or the **SAM** (System Administration Manager) utility. If you use SAM, invoke the **Swap** command within the **Disks and File Systems** menu.

## Maximum data size

To see the current data size limit in the C shell, type:

```
limit datasize
```

The following command displays the current *hard* limit, type:

```
limit -h datasize
```

If the current limit is lower than the hard limit, you can easily raise the current limit. To change the current limit, type:

limit datasize *new_data_size*

If the hard limit is too low you must reconfigure and rebuild the kernel, and then reboot. This is most easily done using **SAM**.

To change **maxdsiz**, use the following path through the SAM menus:

Kernel Configuration → Configurable Parameters → maxdsiz →
Actions → Modify Configurable Parameter →
Specify New Formula/Value → Formula/Value

You can now enter the new maximum data segment size.

You may also need to change the value for **maxdsiz_64**.

Here is the command that lets you rebuild the kernel with these changed values:

Configurable Parameter → Actions → Process New Kernel

Answer **yes** to process the kernel modifications, **yes** to install the new kernel, and **yes** again to reboot the machine with the new kernel.

When the machine reboots, the value you set for **maxdsiz** should be the new hard limit.

## SunOS 5

To find out how much swap space has been allocated and is currently being used, use the **swap** command:

```
% /usr/sbin/swap –s
total: 16192K bytes allocated + 7140K bytes \
           reserved = 23332K used, 63456K available
```

To find out how much swap space is in use while you are running TotalView:

1 Start TotalView with a large executable:

   **totalview** *executable*

2 Press Ctrl-Z to suspend TotalView.

3 Use the following command to see how much swap space TotalView is using:

   /bin/ps –l

   For example, in this case the value in the SZ column is 1036 pages, with each page being 4K in size.

```
F S   UID  PID   PPID  C  PRI NI     ADDR ...
8 T 14694 3456   2558 80    1 20 ff451000 ...
```

To add swap space, use the **mkfile(1M)** and **swap(1M)** commands. You must be root to use these commands. For more information, refer to the online manual pages for these commands.

## IRIX

To find out how much swap space has been allocated and is currently being used, use the **swap** command:

```
% /sbin/swap –s
total: 1.55m allocated + 124.47m add'l reserved =
126.02m bytes used, 250.94m bytes available
```

To find out how much swap space is in use while you are running TotalView:

**1** Start TotalView with a large executable:

**totalview** *executable*

**2** Press Ctrl-Z to suspend TotalView.

**3** Use the following command to see how much swap space TotalView is using:

/bin/ps –l

For example, in this case the value in the SZ column is 584 pages.

```
 F S   UID   PID  PPID  C  PRI NI  P      ...
b0 T 14694 26236 26271  5   62 20  *   ...
```

Use the following command to determine the number of bytes in a page:

sysconf PAGESIZE

To add swap space, use the **mkfile(1M)** and **swap(1M)** commands. You must be root to use these commands. For more information, refer to the online manual pages for these commands.

## Linux

To find out how much swap space has been allocated and is currently being used, use either the **swapon** or **top** commands on Linux:

% /sbin/swapon -s

```
Filename         Type         Size     Used    Priority
/dev/hda7        partition  128484    28      -1
```

% top

```
jcownie@pc2: top
(null) 1:29pm up 4:28, 1 user, load average: 0.00, 0.00, 0.00
52 processes: 50 sleeping, 2 running, 0 zombie, 0 stopped
CPU states: 1.1% user, 0.4% system, 0.0% nice, 98.4% idle
Mem: 127904K a, 116512K used, 11392K free, 36020K shrd, \
            3632K buff
Swap: 128484K av,    28K used, 128456K free       79804K cached
... remainder of "top" listing removed ...
```

You can use the **mkswap**(8) command to create swap space. The **swapon**(8) command tells Linux that it should use this space.

# Shared Libraries

TotalView supports dynamically linked executables, that is, executables that are linked with shared libraries.

When you start TotalView with a dynamically linked executable, TotalView loads an additional set of symbols for the shared libraries, as indicated in the shell from which you started TotalView. To accomplish this, TotalView:

- Runs a sample process and discards it.
- Reads information from the process.
- Reads the symbol table for each library.

When you create a process without starting it, and the process does not include shared libraries, the program counter points to the entry point of the process, usually the **start** routine. If the process does include shared libraries, however, TotalView takes the following actions:

- Runs the dynamic loader (SunOS 5: **ld.so,** Compaq Tru64 UNIX: **/sbin/loader**, Linux: **/lib/ld-linux.so.?**, IRIX: **rld**).
- Sets the PC to point to the location after the invocation of the dynamic loader but before the invocation of C++ static constructors or the **main** routine.

NOTE **ON HP-UX, TotalView cannot stop the loading of shared libraries until after static constructors on shared library initialization routines have been run.**

When you attach to a process that uses shared libraries, TotalView takes the following actions:

- If you attached to the process after the dynamic loader ran, then TotalView loads the dynamic symbols for the shared library.
- If you attached to the process before it runs the dynamic loader, TotalView allows the process to run the dynamic loader to completion. Then, TotalView loads the dynamic symbols for the shared library.

If desired, you can suppress the use of shared libraries by starting TotalView with the **–no_dynamic** option. Refer to Chapter 12 "*TotalView Command Syntax*" on page 299 for details on this TotalView start-up option.

If you believe that a shared library has changed since you started a Totalview session, you can use the **Reload Shared Library Information** command on the **Current/Update/Relative** submenu to reload library symbol tables. Be aware that only some systems such as AIX permit you to reload library information.

### Using Shared Libraries on HP-UX

The dynamic library loader on HP-UX loads shared libraries into shared memory. Writing breakpoints into code sections loaded in shared memory can cause programs not under TotalView's control to fail when they execute an unexpected breakpoint.

If you need to single-step or set breakpoints in shared libraries you must set your application to load those libraries in private memory. This is done using HP's **pxdb** command.

```
pxdb -s on appname  (load shared libraries into private memory)
pxdb -s off appname  (load shared libraries into shared memory)
```

For 64-bit platforms, use **pxdb64** instead of **pxdb**. If the version of **pxdb64.exe** supplied with HP's compilers does not work correctly, you may need to install an HP-supplied patch. You will find additional information on the TOTALVIEW RELEASE NOTES.

## Debugging Dynamically Loaded Libraries

TotalView automatically reads the symbols of shared libraries that are dynamically loaded into your program at runtime. These libraries are ones that are loaded using **dlopen** (or, on IBM AIX, **load** and **loadbind**).

TotalView automatically detects these calls, then loads the symbol table from the newly loaded libraries and plants any enabled saved breakpoints for these libraries. TotalView then decides whether to ask you about stopping the process to plant breakpoints. TotalView decides according to the following rules:

1 If the "**ask on dlopen**" option is set to false, TotalView *does not* ask you about stopping.
2 If one or more of the strings in the "**DLL Do Query on Load**" list is a suffix of the full library name (including path), TotalView asks you about stopping.
3 If one or more of the strings in the "**DLL Don't Query on Load**" list is a prefix of the full library name (including path), TotalView does not ask you about stopping.
4 If the newly loaded libraries have any saved breakpoints, TotalView does not ask you about stopping.
5 If none of the rules above apply, TotalView asks you about stopping.

If TotalView does not ask you about stopping the process, the process is continued.

If TotalView decides to ask you about stopping, it displays a dialog box as shown in Figure 122. To stop the process, answer yes. To allow the process to continue executing, answer no. Stopping the process allows you to insert breakpoints in the newly loaded shared library.


**Figure 122: dlopen Dialog Box**

Control the **–ask_on_dlopen** option by doing either or both of the following:

- Set the command line option **–ask_on_dlopen** to set the "ask on dlopen" option to true, or **–no_ask_on_dlopen** to set it to false.
- Set the X resource "TOTALVIEW*ASKONDLOPEN" on page 276.

You can set the "**DLL Do Query on Load**" and "**DLL Don't Query on Load**" lists initially from the X resources:

- "TOTALVIEW*DLLSTOPSUFFIX" on page 280 sets the "**DLL Do Query on Load**" list and defaults to empty.

- "TOTALVIEW*DLLIGNOREPREFIX" on page 280 sets the "**DLL Don't Query on Load**" list, and defaults to the standard library paths for the system TotalView is running.

The following table lists the "don't query on load list" library paths:

Table 47: Default DLL Don't Query on Load List

| Platform | Value | |
|---|---|---|
| Compaq Tru64 UNIX Alpha | /usr/shlib/ | /usr/ccs/lib/ |
| | /usr/lib/cmplrs/cc/ | /usr/lib/ |
| | /usr/local/lib/ | /var/shlib/ |
| HP-UX | /usr/lib/ | /usr/lib/pa20_64 |
| | /opt/langtools/lib/ | /opt/langtools/lib/pa20_64/ |
| IBM AIX | /lib/ | /usr/lib/ |
| | /usr/lpp/ | /usr/ccs/lib/ |
| | /usr/dt/lib/ | /tmp/ |
| SGI IRIX | /lib/ | /usr/lib/ |
| | /usr/local/lib/ | /lib32/ |
| | /usr/lib32/ | /usr/local/lib32/ |
| | /lib64/ | /usr/lib64/ |
| | /usr/local/lib64 | |
| SUN Solaris 2.x | /lib/ | /usr/lib/ |
| | /usr/ccs/lib/ | |
| Linux x86 | /lib | /usr/lib |
| Linux Alpha | /lib | /usr/lib |

The values of the X resources should be space-separated lists of the prefixes and suffixes to be used. If you change the **dllIgnorePrefix**, you probably want to copy the default values into the new list.

After starting TotalView, you can change these lists using the **Set DLL Do Query on Load** and **Set DLL Don't Query on Load** commands in the **Display/Directory/Edit** submenu in the Process Window.

Breakpoint files written by this version are not, readable by TotalView version 3.8 or earlier if they contain breakpoints in dynamic libraries.

## Known Limitations

Dynamic library support has the following know limitations:

- TotalView doesn’t deal correctly with parallel programs that call **dlopen** on different libraries in different processes. TotalView requires that the processes have a uniform address space, including all shared libraries.
- TotalView doesn’t yet fully support unloading libraries (using **dlclose**) and then reloading them at a different address using **dlopen**.

# Remapping Keys

On the SunOS 5 keyboard you may need to remap the page-up and page-down keys to the Prior and Next keysym so that you can scroll TotalView windows with the page-up and page-down keys. To do so, add the following lines to your X Window System start-up file:

```
# Remap F29/F35 to PgUp/PgDn
xmodmap -e 'keysym F29 = Prior'
xmodmap -e 'keysym F35 = Next'
```

# Expression System

Depending on the target platform, TotalView supports:

- An interpreted expression system only
- Both an interpreted and a compiled expression system

Unless stated otherwise below, TotalView supports interpreted expressions only. See “*Interpreted Versus Compiled Expression Performance*” on page 219 for more information on the differences between interpreted and compiled expressions.

## IBM AIX

On IBM AIX, TotalView supports compiled and interpreted expressions. TotalView also supports assembler in expressions.

Some program functions called from the TotalView expression system on the Power architecture cannot have floating-point arguments which are passed by value. However, in functions with a variable number of arguments, floating-point arguments *can* be in the varying part of the argument list. For example, you can include floating-point arguments with calls to **printf**:

```
double d = 3.14159;
printf("d = %f\n", d);
```

## Compaq Tru64 UNIX

On Compaq Tru64 UNIX, TotalView supports compiled and interpreted expressions. TotalView also supports assembler in expressions.

## SGI IRIX

On IRIX, TotalView supports compiled and interpreted expressions. TotalView also supports assembler in expressions.

TotalView includes the SGI IRIX expression compiler. This feature does not use any MIPS-IV specific instructions. It does use MIPS-III instructions freely. It fully supports –n32 and –64 executables.

Due to limitations when dynamically allocating patch space, compiled expression are disabled by default on SGI IRIX. To enable compiled expressions in an invocation of TotalView, use the X resource "TOTALVIEW*COMPILEEXPRESSIONS" on page 279 to set the option to true, or pass the X resource as the command line option **–compileExpressions=true**. This option also tells TotalView to find or allocate patch space in your program for code fragments generated by the expression compiler.

If you enable compiled patches on SGI IRIX with a multiprocess program, you must use static patches. For example, if you link a static patch space into an IRIX MPI program and run the program under TotalView's control, TotalView should let you debug it. If you attach to a previously started MPI job, however, even static patches won't let the program run properly. If TotalView still fails to work properly with the static patch space, then you probably can't use compiled patches with your program.

For general instructions on using patch space allocation controls with compiled expressions, see "*Allocating Patch Space for Compiled Expressions*" on page 220.

Appendix C

# Architectures

This appendix describes the architectures TotalView supports, including:

- Power
- Alpha
- HP PA-RISC
- SPARC
- MIPS
- Intel-x86 (Intel 80386, 80486 and Pentium processors)

It includes the following topics for each architecture:

- General registers
- Floating-point registers
- Floating-point format

## Power

### Power General Registers

TotalView displays Power general registers in the Stack Frame Pane of the Process Window. The following table describes how TotalView treats each general register, and the actions you can take with each register.

Table 48: Power General Purpose Integer Registers

| Register | Description | Data Type | Edit | Dive | Specify in Expression |
|---|---|---|---|---|---|
| R0 | General register 0 | <int> | yes | yes | $r0 |
| SP | Stack pointer | <int> | yes | yes | $sp |

Table 48: Power General Purpose Integer Registers (cont.)

| Register | Description | Data Type | Edit | Dive | Specify in Expression |
|---|---|---|---|---|---|
| RTOC | TOC pointer | <int> | yes | yes | $rtoc |
| R3 – R31 | General registers 3 – 31 | <int> | yes | yes | $r3 – $r31 |
| INUM | | <int> | yes | no | $inum |
| PC | Program counter | <code>[] | no | yes | $pc |
| SRR1 | Machine status save/restore register | <int> | yes | no | $srr1 |
| LR | Link register | <int> | yes | no | $lr |
| CTR | Counter register | <int> | yes | no | $ctr |
| CR | Condition register | <int> | yes | no | $cr |
| XER | Integer exception register | <int> | yes | no | $xer |
| DAR | Data address register | <int> | yes | no | $dar |
| MQ | MQ register | <int> | yes | no | $mq |
| MSR | Machine state register | <int> | yes | no | $msr |
| SEG0 – SEG9 | Segment registers 0 – 9 | <int> | yes | no | $seg0 – $seg9 |
| SG10 – SG15 | Segment registers 10 –15 | <int> | yes | no | $sg10 – $sg15 |
| SCNT | SS_COUNT | <int> | yes | no | $scnt |
| SAD1 | SS_ADDR 1 | <int> | yes | no | $sad1 |
| SAD2 | SS_ADDR 2 | <int> | yes | no | $sad2 |
| SCD1 | SS_CODE 1 | <int> | yes | no | $scd1 |
| SCD2 | SS_CODE 2 | <int> | yes | no | $scd2 |
| TID | | <int> | yes | no | |

## Power MSR Register

For your convenience, TotalView interprets the bit settings of the Power MSR register. You can edit the value of the MSR and set it to any of the bit settings outlined in the following table.

Table 49: Power MSR Register Bit Settings

| Value | Bit Setting | Meaning |
|---|---|---|
| 0x00040000 | POW | Power management enable |
| 0x00020000 | TGPR | Temporary GPR mapping |

Table 49: Power MSR Register Bit Settings (cont.)

| Value | Bit Setting | Meaning |
|---|---|---|
| 0x00010000 | ILE | Exception little-endian mode |
| 0x00008000 | EE | External interrupt enable |
| 0x00004000 | PR | Privilege level |
| 0x00002000 | FP | Floating-point available |
| 0x00001000 | ME | Machine check enable |
| 0x00000800 | FE0 | Floating-point exception mode 0 |
| 0x00000400 | SE | Single-step trace enable |
| 0x00000200 | BE | Branch trace enable |
| 0x00000100 | FE1 | Floating-point exception mode 1 |
| 0x00000040 | IP | Exception prefix |
| 0x00000020 | IR | Instruction address translation |
| 0x00000010 | DR | Data address translation |
| 0x00000002 | RI | Recoverable exception |
| 0x00000001 | LE | Little-endian mode enable |

## Power Floating-Point Registers

TotalView displays the Power floating-point registers in the Stack Frame Pane of the Process Window. The next table describes how TotalView treats each floating-point register, and the actions you can take with each register.

Table 50: Power Floating-Point Registers

| Register | Description | Data Type | Edit | Dive | Specify in Expression |
|---|---|---|---|---|---|
| F0 – F31 | Floating-point registers 0 – 31 | <double> | yes | yes | $f0 – $f31 |
| FPSCR | Floating-point status register | <int> | yes | no | $fpscr |
| FPSCR2 | Floating-point status register 2 | <int> | yes | no | $fpscr2 |

## Power FPSCR Register

For your convenience, TotalView interprets the bit settings of the Power FPSCR register. You can edit the value of the FPSCR and set it to any of the bit settings outlined in the following table.

Table 51: Power PFSCR Register Bit Settings

| Value | Bit Setting | Meaning |
|---|---|---|
| 0x80000000 | FX | Floating-point exception summary |
| 0x40000000 | FEX | Floating-point enabled exception summary |
| 0x20000000 | VX | Floating-point invalid operation exception summary |
| 0x10000000 | OX | Floating-point overflow exception |
| 0x08000000 | UX | Floating-point underflow exception |
| 0x04000000 | ZX | Floating-point zero divide exception |
| 0x02000000 | XX | Floating-point inexact exception |
| 0x01000000 | VXSNAN | Floating-point invalid operation exception for SNaN |
| 0x00800000 | VXISI | Floating-point invalid operation exception: $\infty - \infty$ |
| 0x00400000 | VXIDI | Floating-point invalid operation exception: $\infty / \infty$ |
| 0x00200000 | VXZDZ | Floating-point invalid operation exception: 0 / 0 |
| 0x00100000 | VXIMZ | Floating-point invalid operation exception: $\infty * \infty$ |
| 0x00080000 | VXVC | Floating-point invalid operation exception: invalid compare |
| 0x00040000 | FR | Floating-point fraction rounded |
| 0x00020000 | FI | Floating-point fraction inexact |
| 0x00010000 | FPRF=(C) | Floating-point result class descriptor |
| 0x00008000 | FPRF=(L) | Floating-point less than or negative |
| 0x00004000 | FPRF=(G) | Floating-point greater than or positive |
| 0x00002000 | FPRF=(E) | Floating-point equal or zero |
| 0x00001000 | FPRF=(U) | Floating-point unordered or NaN |
| 0x00011000 | FPRF=(QNAN) | Quiet NaN; alias for FPRF=(C+U) |
| 0x00009000 | FPRF=(-INF) | -Infinity; alias for FPRF=(L+U) |
| 0x00008000 | FPRF=(-NORM) | -Normalized number; alias for FPRF=(L) |

Table 51: Power PFSCR Register Bit Settings (cont.)

| Value | Bit Setting | Meaning |
|---|---|---|
| 0x00018000 | FPRF=(-DENORM) | -Denormalized number; alias for FPRF=(C+L) |
| 0x00012000 | FPRF=(-ZERO) | -Zero; alias for FPRF=(C+E) |
| 0x00002000 | FPRF=(+ZERO) | +Zero; alias for FPRF=(E) |
| 0x00014000 | FPRF=(+DENORM) | +Denormalized number; alias for FPRF=(C+G) |
| 0x00004000 | FPRF=(+NORM) | +Normalized number; alias for FPRF=(G) |
| 0x00005000 | FPRF=(+INF) | +Infinity; alias for FPRF=(G+U) |
| 0x00000400 | VXSOFT | Floating-point invalid operation exception: software request |
| 0x00000200 | VXSQRT | Floating-point invalid operation exception: square root |
| 0x00000100 | VXCVI | Floating-point invalid operation exception: invalid integer convert |
| 0x00000080 | VE | Floating-point invalid operation exception enable |
| 0x00000040 | OE | Floating-point overflow exception enable |
| 0x00000020 | UE | Floating-point underflow exception enable |
| 0x00000010 | ZE | Floating-point zero divide exception enable |
| 0x00000008 | XE | Floating-point inexact exception enable |
| 0x00000004 | NI | Floating-point non-IEEE mode enable |
| 0x00000000 | RN=NEAR | Round to nearest |
| 0x00000001 | RN=ZERO | Round toward zero |
| 0x00000002 | RN=PINF | Round toward +infinity |
| 0x00000003 | RN=NINF | Round toward -infinity |

## Using the Power FPSCR Register

On AIX, if you compile your program to catch floating point exceptions (IBM compiler -qflttrap option), you can change the value of the FPSCR within TotalView to customize the exception handling for your program.

For example, if your program inadvertently divides by zero, you can edit the bit setting of the FPSCR register in the Stack Frame Pane. In this case, you would change the bit setting for the FPSCR to include 0x10 (as shown in Table 51) so that TotalView traps the “divide by zero” exception. The string displayed next to the FPSR register should now include “ZE”. Now, when your program divides by zero, it receives a SIGTRAP signal, which will be caught by TotalView. See Chapter 3 "*Setting Up a Debugging Session*" on page 29 and "*Handling Signals*" on page 41 for more information. If you did not set the bit for trapping divide by zero or you did not compile to catch floating point exceptions, your program would not stop and the processor would set the "ZX" bit.

### Power Floating-Point Format

The Power architecture supports the IEEE floating-point format.

## HP PA-RISC

### PA-RISC General Registers

TotalView displays the PA-RISC general registers in the Stack Frame Pane of the Process Window. The following table describes how TotalView treats each general register and the actions you take with them.

Table 52: PA-RISC General Registers

| Register | Description | Data Type | Edit | Dive | Specify in Expression |
|---|---|---|---|---|---|
| r0 | always contains zero | <long> | no | no | $r0 |
| r1-r31 | general registers | <long> | yes | yes | $r1-$r31 |
| pc | current instruction pointer | <long> | yes | yes | $pc |
| nxtpc | next instruction pointer | <long> | yes | yes | $nxtpc |
| pcs | current instruction space | <long> | no | no | $pcs |
| nxtpcs | next instruction space | <long> | no | no | $nxtpcs |
| psw | processor status word | <long> | yes | no | $psw |
| sar | shift amount register | <long> | yes | no | $sar |
| sr0-sr7 | space registers | <long> | no | no | $sr0-$sr7 |
| recov | recovery counter | <long> | no | no | $recov |
| pid1-pid8 | protection ids | <long> | no | no | $pid1-$pid8 |
| ccr | coprocessor configuration | <long> | no | no | $ccr |
| scr | SFU configuration register | <long> | no | no | $scr |

Table 52: PA-RISC General Registers

| Register | Description | Data Type | Edit | Dive | Specify in Expression |
|---|---|---|---|---|---|
| eiem | external interrupt enable mask | <long> | no | no | $eiem |
| iir | interrupt instruction | <long> | no | no | $iir |
| isr | interrupt space | <long> | no | no | $isr |
| ior | interrupt offset | <long> | no | no | $ior |
| cr24-cr26 | temporary registers | <long> | no | no | $cr24-$cr26 |
| tp | thread pointer | <long> | yes | yes | $tp |

## PA-RISC Process Status Word

For your convenience, TotalView interprets the bit settings of the PA-RISC Processor Status Word. You can edit the value of this word and set some of the bits listed in the following table

Table 53: PA-RISC Processor Status Word

| Value | Bit Setting | Meaning |
|---|---|---|
| W | 0x0000000008000000 | 64-bit addressing enable |
| E | 0x0000000004000000 | little-endian enable |
| S | 0x0000000002000000 | secure interval timer |
| T | 0x0000000001000000 | taken branch flag |
| H | 0x0000000000800000 | higher privilege transfer trap enable |
| L | 0x0000000000400000 | lower privilege transfer trap enable |
| N | 0x0000000000200000 | nullify current instruction |
| X | 0x0000000000100000 | data memory break disable |
| B | 0x0000000000080000 | taken branch flag |
| C | 0x0000000000040000 | code address translation enable |
| V | 0x0000000000020000 | divide step correction |
| M | 0x0000000000010000 | high-priority machine check mask |
| O | 0x0000000000000080 | ordered references |
| F | 0x0000000000000020 | performance monitor interrupt unmask |
| R | 0x0000000000000010 | recovery counter enable |
| Q | 0x0000000000000008 | interrupt state collection enable |
| P | 0x0000000000000004 | protection identifier validation enable |

Table 53: PA-RISC Processor Status Word (cont.)

| Value | Bit Setting | Meaning |
|---|---|---|
| D | 0x0000000000000002 | data address translation enable |
| I | 0x0000000000000001 | external interrupt unmask |
| C/B | 0x000000FF0000FF00 | carry/borrow bits |

## PA-RISC Floating-Point Registers

The PA-RISC has 32 floating point registers. The first four are used for status and exception registers. The rest can be addressed as 64 bit doubles, as two 32 bit floats in the right and left sides of the register, or even-odd pairs of registers as 128 bit extended floats.

Table 54: PA-RISC Floating-Point Registers

| Register | Description | Data Type | Edit | Dive | Specify in Expression |
|---|---|---|---|---|---|
| status | Status register | <int> | no | no | $status |
| er1-er7 | Exception registers | <int> | no | no | $er1-$er7 |
| fr4-fr31 | Double floating point registers | <double> | yes | yes | $fr4-$fr31 |
| fr4l-fr31l | Left half floating point registers | <float> | yes | yes | $fr4l-$fr31l |
| fr4r-fr31r | Right half floating point registers | <float> | yes | yes | $fr4r-$fr31r |
| fr4/fr5-fr30/fr31 | Extended floating point register pairs | <extended> | yes | yes | $fr4_fr5-$fr30_fr31 |

The floating-point status word controls the arithmetic rounding mode, enables user-level traps, enables floating point exceptions, and indicates the results of comparisons.

Table 55: Floating-Point Status Word Use

| Type | Value | Meaning |
|---|---|---|
| Rounding Mode | 0 | Round to nearest |
| | 1 | Round towards zero |
| | 2 | Round towards +infinity |
| | 3 | Round towards -infinity |

Table 55: Floating-Point Status Word Use (cont.)

| Type | Value | Meaning |
|---|---|---|
| Exception Enable and Exception Flag Bits | V | Invalid operation |
| | Z | Division by zero |
| | O | Overflow |
| | U | Underflow |
| | I | Inexact result |
| Comparison Fields | C | Compare bit, contains the result of the most recent queued compare instruction |
| | CQ | Compare queue, contains the result of the second-most recent queued compare through the twelfth-most recent queued compare; each queued compare instruction shifts the CQ field right one bit and copies the C bit into the left most position<br>This field occupies the same bits as the CA field and is undefined after a targeted compare |
| | CA | Compare array, is an array of seven compare bits, each of which contains the result of the most recent compare instruction targeting that bit<br>This field occupies the same bits as the CQ field and is undefined after a queued compare |
| Other Flags: | T | Delayed trap |
| | D | Denormalized as zero |

## PA-RISC Floating-Point Format

The PA-RISC processor supports the IEEE floating-point format.

# SPARC

## SPARC General Registers

TotalView displays the SPARC general registers in the Stack Frame Pane of the Process Window. The following table describes how TotalView treats each general register, and the actions you can take with each register.

Table 56: SPARC General Registers

| Register | Description | Data Type | Edit | Dive | Specify in Expression |
|---|---|---|---|---|---|
| G0 | Global zero register | <int> | no | no | $g0 |
| G1 – G7 | Global registers | <int> | yes | yes | $g1 – $g7 |
| O0 – O5 | Outgoing parameter registers | <int> | yes | yes | $o0 – $o5 |
| SP | Stack pointer | <int> | yes | yes | $sp |
| O7 | Temporary register | <int> | yes | yes | $o7 |
| L0 – L7 | Local registers | <int> | yes | yes | $l0 – $l7 |
| I0 – I5 | Incoming parameter registers | <int> | yes | yes | $i0 – $i5 |
| FP | Frame pointer | <int> | yes | yes | $fp |
| I7 | Return address | <int> | yes | yes | $i7 |
| PSR | Processor status register | <int> | yes | no | $psr |
| Y | Y register | <int> | yes | yes | $y |
| WIM | WIM register | <int> | no | no | |
| TBR | TBR register | <int> | no | no | |
| PC | Program counter | <code>[] | no | yes | $pc |
| nPC | Next program counter | <code>[] | no | yes | $npc |

## SPARC PSR Register

For your convenience, TotalView interprets the bit settings of the SPARC PSR register. You can edit the value of the PSR and set some of the bits outlined in the following table.

Table 57: SPARC PSR Register Bit Settings

| Value | Bit Setting | Meaning |
|---|---|---|
| ET | 0x00000020 | Traps enabled |
| PS | 0x00000040 | Previous supervisor |
| S | 0x00000080 | Supervisor mode |
| EF | 0x00001000 | Floating-point unit enabled |
| EC | 0x00002000 | Coprocessor enabled |
| C | 0x00100000 | Carry condition code |
| V | 0x00200000 | Overflow condition code |
| Z | 0x00400000 | Zero condition code |
| N | 0x00800000 | Negative condition code |

## SPARC Floating-Point Registers

TotalView displays the SPARC floating-point registers in the Stack Frame Pane of the Process Window. The next table describes how TotalView treats each floating-point register, and the actions you can take with each register.

Table 58: SPARC Floating-Point Registers

| Register | Description | Data Type | Edit | Dive | Specify in Expression |
|---|---|---|---|---|---|
| F0 – F31 | Floating-point registers (*f* registers), used singly | <float> | yes | yes | $f0 – $f31 |
| F0/F1 – F30/F31 | Floating point registers (*f* registers), used as pairs | <double> | yes | yes | $f0_f1 – $f30_f31 |
| FPCR | Floating-point control register | <int> | no | no | $fpcr |
| FPSR | Floating-point status register | <int> | yes | no | $fpsr |

TotalView allows you to use these registers singly or in pairs, depending on how they are used by your program. For example, if you use F1 by itself, its type is <float>, but if you use the F0/F1 pair, its type is <double>.

## SPARC FPSR Register

For your convenience, TotalView interprets the bit settings of the SPARC FPSR register. You can edit the value of the FPSR and set it to any of the bit settings outlined in the following table.

Table 59: SPARC FPSR Register Bit Settings

| Value | Bit Setting | Meaning |
|---|---|---|
| CEXC=NX | 0x00000001 | Current inexact exception |
| CEXC=DZ | 0x00000002 | Current divide by zero exception |
| CEXC=UF | 0x00000004 | Current underflow exception |
| CEXC=OF | 0x00000008 | Current overflow exception |
| CEXC=NV | 0x00000010 | Current invalid exception |
| AEXC=NX | 0x00000020 | Accrued inexact exception |
| AEXC=DZ | 0x00000040 | Accrued divide by zero exception |
| AEXC=UF | 0x00000080 | Accrued underflow exception |
| AEXC=OF | 0x00000100 | Accrued overflow exception |
| AEXC=NV | 0x00000200 | Accrued invalid exception |
| EQ | 0x00000000 | Floating-point condition = |
| LT | 0x00000400 | Floating-point condition < |
| GT | 0x00000800 | Floating-point condition > |
| UN | 0x00000c00 | Floating-point condition unordered |
| QNE | 0x00002000 | Queue not empty |
| NONE | 0x00000000 | Floating-point trap type None |
| IEEE | 0x00004000 | Floating-point trap type IEEE Exception |
| UFIN | 0x00008000 | Floating-point trap type Unfinished FPop |
| UIMP | 0x0000c000 | Floating-point trap type Unimplemented FPop |
| SEQE | 0x00010000 | Floating-point trap type Sequence Error |
| NS | 0x00400000 | Non-standard floating-point FAST mode |
| TEM=NX | 0x00800000 | Trap enable mask – Inexact Trap Mask |
| TEM=DZ | 0x01000000 | Trap enable mask – Divide by Zero Trap Mask |
| TEM=UF | 0x02000000 | Trap enable mask – Underflow Trap Mask |

Table 59: SPARC FPSR Register Bit Settings (cont.)

| Value | Bit Setting | Meaning |
|---|---|---|
| TEM=OF | 0x04000000 | Trap enable mask – Overflow Trap Mask |
| TEM=NV | 0x08000000 | Trap enable mask – Invalid Operation Trap Mask |
| EXT | 0x00000000 | Extended rounding precision – Extended precision |
| SGL | 0x10000000 | Extended rounding precision – Single precision |
| DBL | 0x20000000 | Extended rounding precision – Double precision |
| NEAR | 0x00000000 | Rounding direction – Round to nearest (tie-even) |
| ZERO | 0x40000000 | Rounding direction – Round to 0 |
| PINF | 0x80000000 | Rounding direction – Round to +Infinity |
| NINF | 0xc0000000 | Rounding direction – Round to -Infinity |

## Using the SPARC FPSR Register

The SPARC processor does not catch floating-point errors by default. You can change the value of the FPSR within TotalView to customize the exception handling for your program.

For example, if your program inadvertently divides by zero, you can edit the bit setting of the FPSR register in the Stack Frame Pane. In this case, you would change the bit setting for the FPSR to include 0x01000000 (as shown in Table 59) so that TotalView traps the "divide by zero" bit. The string displayed next to the FPSR register should now include TEM=(DZ). Now, when your program divides by zero, it receives a SIGFPE signal, which you can catch with TotalView. See Chapter 3 "*Setting Up a Debugging Session*" on page 29 and "*Handling Signals*" on page 41 for more information. If you did not set the bit for trapping divide by zero, the processor would ignore the error and set the AEXC=(DZ) bit.

## SPARC Floating-Point Format

The SPARC processor supports the IEEE floating-point format.

# Alpha

## Alpha General Registers

TotalView displays the Alpha general registers in the Stack Frame Pane of the Process Window. The next table describes how TotalView treats each general register, and the actions you can take with each register.

Table 60: Alpha General Purpose Integer Registers

| Register | Description | Data Type | Edit | Dive | Specify in Expression |
|---|---|---|---|---|---|
| V0 | Function value register | <long> | yes | yes | $v0 |
| T0 – T7 | Conventional scratch registers | <long> | yes | yes | $t0 – $t7 |
| S0 – S5 | Conventional saved registers | <long> | yes | yes | $s0 – $s5 |
| S6 | Stack frame base register | <long> | yes | yes | $s6 |
| A0 – A5 | Argument registers | <long> | yes | yes | $a0 – $a5 |
| T8 – T11 | Conventional scratch registers | <long> | yes | yes | $t8 – $t11 |
| RA | Return Address register | <long> | yes | yes | $ra |
| T12 | Procedure value register | <long> | yes | yes | $t12 |
| AT | Volatile scratch register | <long> | yes | yes | $at |
| GP | Global pointer register | <long> | yes | yes | $gp |
| SP | Stack pointer | <long> | yes | yes | $sp |
| ZERO | ReadAsZero/Sink register | <long> | no | yes | $zero |
| PC | Program counter | <code>[] | no | yes | $pc |
| FP | Frame pointer; the Frame Pointer (FP) is a software register that TotalView maintains; it is not an actual hardware register—TotalView computes the value of FP as part of the stack backtrace | <long> | no | yes | $fp |

## Alpha Floating-Point Registers

TotalView displays the Alpha floating-point registers in the Stack Frame Pane of the Process Window. Here is a table describes how TotalView treats each floating-point register, and the actions you can take with each register.

Table 61: Alpha Floating-Point Registers

| Register | Description | Data Type | Edit | Dive | Specify in Expression |
|---|---|---|---|---|---|
| F0 – F1 | Floating-point registers (*f* registers), used singly | <double> | yes | yes | $f0 – $f1 |
| F2 – F9 | Conventional saved registers | <double> | yes | yes | $f2 – $f9 |
| F10 – F15 | Conventional scratch registers | <double> | yes | yes | $f10 – $f15 |
| F16 – F21 | Argument registers | <double> | yes | yes | $f16 – $f21 |
| F22 – F30 | Conventional scratch registers | <double> | yes | yes | $f22 – $f30 |
| F31 | ReadAsZero/Sink register | <double> | yes | yes | $f31 |
| FPCR | Floating-point control register | <long> | yes | no | $fpcr |

## Alpha FPCR Register

For your convenience, TotalView interprets the bit settings of the Alpha FPCR register. You can edit the value of the FPCR and set it to any of the bit settings outlined in the following table.

Alpha FPCR Register Bit Settings

| Value | Bit Setting | Meaning |
|---|---|---|
| SUM | 0x8000000000000000 | Summary bit |
| DYN=CHOP | 0x0000000000000000 | Rounding mode — Chopped rounding mode |
| DYN=MINF | 0x0400000000000000 | Rounding mode — Minus infinity |

| Value | Bit Setting | Meaning |
|---|---|---|
| DYN=NORM | 0x0800000000000000 | Rounding mode — Normal rounding |
| DYN=PINF | 0x0c00000000000000 | Rounding mode — Plus infinity |
| IOV | 0x0200000000000000 | Integer overflow |
| INE | 0x0100000000000000 | Inexact result |
| UNF | 0x0080000000000000 | Underflow |
| OVF | 0x0040000000000000 | Overflow |
| DZE | 0x0020000000000000 | Division by zero |
| INV | 0x0010000000000000 | Invalid operation |

### Alpha Floating-Point Format

The Alpha processor supports the IEEE floating point format.

## MIPS

### MIPS General Registers

TotalView displays the MIPS general purpose registers in the Stack Frame Pane of the Process Window. The following table describes how TotalView treats each general register, and the actions you can take with each register.

Programs compiled either **–64** or **–n32** have 64 bit registers. TotalView uses **<long>** for **–64** compiled programs and **<long long>** for **–n32** compiled programs.

**Table 62: MIPS General (Integer) Registers**

| Register | Description | Data Type | Edit | Dive | Specify in Expression |
|---|---|---|---|---|---|
| ZERO | Always has the value 0 | <long> | no | no | $zero |
| AT | Reserved for the assembler | <long> | yes | yes | $at |
| V0 – V1 | Function value registers | <long> | yes | yes | $v0 – $v1 |

Table 62: MIPS General (Integer) Registers (cont.)

| Register | Description | Data Type | Edit | Dive | Specify in Expression |
|---|---|---|---|---|---|
| A0 – A7 | Argument registers | <long> | yes | yes | $a0 – $a7 |
| T0 – T3 | Temporary registers | <long> | yes | yes | $t0 – $t3 |
| S0 – S7 | Saved registers | <long> | yes | yes | $s0 – $s7 |
| T8 – T9 | Temporary registers | <long> | yes | yes | $t8 – $t9 |
| K0 – K1 | Reserved for the operating system | <long> | yes | yes | $k1 – $k2 |
| GP | Global pointer | <long> | yes | yes | $gp |
| SP | Stack pointer | <long> | yes | yes | $sp |
| S8 | Hardware frame pointer | <long> | yes | yes | $s8 |
| RA | Return address register | <code>[] | no | yes | $ra |
| MDLO | Multiply/Divide special register, holds least-significant bits of multiply, quotient of divide | <long> | yes | yes | $mdlo |
| MDHI | Multiply/Divide special register, holds most-significant bits of multiply, remainder of divide | <long> | yes | yes | $mdhi |
| CAUSE | Cause register | <long> | yes | yes | $cause |
| EPC | Program counter | <code>[] | no | yes | $epc |
| SR | Status register | <long> | no | no | $sr |
| VFP | Virtual frame pointer<br>The virtual frame pointer is a software register that TotalView maintains. It is not an actual hardware register. TotalView computes the VFP as part of stack backtrace. | <long> | no | no | $vfp |

## MIPS SR Register

For your convenience, TotalView interprets the bit settings of the SR register as outlined in the next table.

Table 63: MIPS SR Register Bit Settings

| Value | Bit Setting | Meaning |
|---|---|---|
| 0x00000001 | IE | Interrupt enable |
| 0x00000002 | EXL | Exception level |
| 0x00000004 | ERL | Error level |
| 0x00000008 | S | Supervisor mode |
| 0x00000010 | U | User mode |
| 0x00000018 | U | Undefined (implemented as User mode) |
| 0x00000000 | K | Kernel mode |
| 0x00000020 | UX | User mode 64-bit addressing |
| 0x00000040 | SX | Supervisor mode 64-bit addressing |
| 0x00000080 | KX | Kernel mode 64-bit addressing |
| 0x0000FF00 | IM=*i* | Interrupt Mask value is *i* |
| 0x00010000 | DE | Disable cache parity/ECC |
| 0x00020000 | CE | Reserved |
| 0x00040000 | CH | Cache hit |
| 0x00080000 | NMI | Non-maskable interrupt has occurred |
| 0x00100000 | SR | Soft reset or NMI exception |
| 0x00200000 | TS | TLB shutdown has occurred |
| 0x00400000 | BEV | Bootstrap vectors |
| 0x02000000 | RE | Reverse-Endian bit |
| 0x04000000 | FR | Additional floating-point registers enabled |
| 0x08000000 | RP | Reduced power mode |
| 0x10000000 | CU0 | Coprocessor 0 usable |
| 0x20000000 | CU1 | Coprocessor 1 usable |
| 0x40000000 | CU2 | Coprocessor 2 usable |
| 0x80000000 | XX | MIPS IV instructions usable |

## MIPS Floating-Point Registers

TotalView displays the MIPS floating-point registers in the Stack Frame Pane of the Process Window. Here is a table that describes how TotalView treats each floating-point register, and the actions you can take with each register.

Table 64: MIPS Floating-Point Registers

| Register | Description | Data Type | Edit | Dive | Specify in Expression |
|---|---|---|---|---|---|
| F0, F2 | Hold results of floating-point type function. $f0 has the real part, $f2 has the imaginary part | <double> | yes | yes | $f0, $f2 |
| F1 – F3, F4 – F11 | Temporary registers | <double> | yes | yes | $f1 – $f3, $f4 – $f11 |
| F12 – F19 | Pass single or double precision actual arguments | <double> | yes | yes | $f12 – $f19 |
| F20 – F23 | Temporary registers | <double> | yes | yes | $f20 – $f23 |
| F24 – F31 | Saved registers | <double> | yes | yes | $f24 – $f31 |
| FCSR | FPU control and status register | <int> | yes | no | $fcsr |

## MIPS FCSR Register

For your convenience, TotalView interprets the bit settings of the MIPS FCSR register. You can edit the value of the FCSR and set it to any of the bit settings outlined in the following table

Table 65: MIPS FCSR Register Bit Settings

| Value | Bit Setting | Meaning |
|---|---|---|
| RM=RN | 0x00000000 | Round to nearest |
| RM=RZ | 0x00000001 | Round toward zero |
| RM=RP | 0x00000002 | Round toward plus infinity |
| RM=RM | 0x00000003 | Round toward minus infinity |
| flags=(I) | 0x00000004 | Flag=inexact result |
| flags=(U) | 0x00000008 | Flag=underflow |
| flags=(O) | 0x00000010 | Flag=overflow |
| flags=(Z) | 0x00000020 | Flag=divide by zero |

Table 65: MIPS FCSR Register Bit Settings (cont.)

| Value | Bit Setting | Meaning |
|---|---|---|
| flags=(V) | 0x00000040 | Flag=invalid operation |
| enables=(I) | 0x00000080 | Enables=inexact result |
| enables=(U) | 0x00000100 | Enables=underflow |
| enables=(O) | 0x00000200 | Enables=overflow |
| enables=(Z) | 0x00000400 | Enables=divide by zero |
| enables=(V) | 0x00000800 | Enables=invalid operation |
| cause=(I) | 0x00001000 | Cause=inexact result |
| cause=(U) | 0x00002000 | Cause=underflow |
| cause=(O) | 0x00004000 | Cause=overflow |
| cause=(Z) | 0x00008000 | Cause=divide by zero |
| cause=(V) | 0x00010000 | Cause=invalid operation |
| cause=(E) | 0x00020000 | Cause=unimplemented |
| FCC=(0/c) | 0x00800000 | FCC=Floating-Point Condition Code 0; c=Condition bit |
| FS | 0x01000000 | Flush to zero |
| FCC=(1) | 0x02000000 | FCC=Floating-Point Condition Code 1 |
| FCC=(2) | 0x04000000 | FCC=Floating-Point Condition Code 2 |
| FCC=(3) | 0x08000000 | FCC=Floating-Point Condition Code 3 |
| FCC=(4) | 0x10000000 | FCC=Floating-Point Condition Code 4 |
| FCC=(5) | 0x20000000 | FCC=Floating-Point Condition Code 5 |
| FCC=(6) | 0x40000000 | FCC=Floating-Point Condition Code 6 |
| FCC=(7) | 0x80000000 | FCC=Floating-Point Condition Code 7 |

## Using the MIPS FCSR Register

You can change the value of the MIPS FCSR register within TotalView to customize the exception handling for your program.

For example, if your program inadvertently divides by zero, you can edit the bit setting of the FCSR register in the Stack Frame Pane. In this case, you would change the bit setting for the FCSR to include 0x400 (as shown in Table 65). The string displayed next to the FCSR register should now include "enables=(Z)". Now, when your program divides by zero, it receives a SIGFPE signal, which you can catch with TotalView. See Chapter 3 "*Setting Up a*

*Debugging Session*" on page 29 and "*Handling Signals*" on page 41 for more information.

## MIPS Floating-Point Format

The MIPS processor supports the IEEE floating point format.

## MIPS Delay Slot Instructions

On the MIPS architecture, jump and branch instructions have a "delay slot". This means that the instruction after the jump or branch instruction is executed before the jump or branch is executed.

In addition, there is a group of "branch likely" conditional branch instructions in which the instruction in the delay slot is executed only if the branch is taken.

The MIPS processors execute the jump or branch instruction and the delay slot instruction as an indivisible unit. If an exception occurs as a result of executing the delay slot instruction, the branch or jump instruction is not executed, and the exception appears to have been caused by the jump or branch instruction.

This behavior of the MIPS processors affects both the TotalView instruction step command and TotalView breakpoints.

The TotalView instruction step command will step both the jump or branch instruction and the delay slot instruction as if they were a single instruction.

If a breakpoint is placed on a delay slot instruction, execution will stop at the jump or branch preceding the delay slot instruction, and TotalView will not know that it is at a breakpoint. At this point, attempting to continue the thread which hit the breakpoint without first removing the breakpoint will cause the thread to hit the breakpoint again without executing any instructions. Before continuing the thread, you must remove the breakpoint. If you need to reestablish the breakpoint, you might then use the instruction step command to execute just the delay slot instruction and the branch.

A breakpoint placed on a delay slot instruction of a "branch likely" instruction will be hit only if the branch is going to be taken.

# Intel-x86

## Intel-x86 General Registers

TotalView displays the Intel-x86 general registers in the Stack Frame Pane of the Process Window. The following table describes how TotalView treats each general register, and the actions you can take with each register.

Table 66: Intel-x86 General Registers

| Register | Description | Data Type | Edit | Dive | Specify in Expression |
|---|---|---|---|---|---|
| EAX | General registers | <void> | yes | yes | $eax |
| ECX | | <void> | yes | yes | $ecx |
| EDX | | <void> | yes | yes | $edx |
| EBX | | <void> | yes | yes | $ebx |
| EBP | | <void> | yes | yes | $ebp |
| ESP | | <void> | yes | yes | $esp |
| ESI | | <void> | yes | yes | $esi |
| EDI | | <void> | yes | yes | $edi |
| CS | Selector registers | <void> | no | no | $cs |
| SS | | <void> | no | no | $ss |
| DS | | <void> | no | no | $ds |
| ES | | <void> | no | no | $es |
| FS | | <void> | no | no | $fs |
| GS | | <void> | no | no | $gs |
| EFLAGS | | <void> | no | no | $eflags |
| EIP | Instruction pointer | <code>[] | no | yes | $eip |
| FAULT | | <void> | no | no | $fault |
| TEMP | | <void> | no | no | $temp |
| INUM | | <void> | no | no | $inum |
| ECODE | | <void> | no | no | $ecode |

## Intel-x86 Floating-Point Registers

TotalView displays the x86 floating-point registers in the Stack Frame Pane of the Process Window. The next table describes how TotalView treats each floating-point register, and the actions you can take with each register.

Table 67: Intel-x86 Floating-Point Registers

| Register | Description | Data Type | Edit | Dive | Specify in Expression |
|---|---|---|---|---|---|
| ST0 | ST(0) | <extended> | yes | yes | $st0 |
| ST1 | ST(1) | <extended> | yes | yes | $st1 |
| ST2 | ST(2) | <extended> | yes | yes | $st2 |
| ST3 | ST(3) | <extended> | yes | yes | $st3 |
| ST4 | ST(4) | <extended> | yes | yes | $st4 |
| ST5 | ST(5) | <extended> | yes | yes | $st5 |
| ST6 | ST(6) | <extended> | yes | yes | $st6 |
| ST7 | ST(7) | <extended> | yes | yes | $st7 |
| FPCR | Floating-point control register | <void> | yes | no | $fpcr |
| FPSR | Floating-point status register | <void> | no | no | $fpsr |
| FPTAG | Tag word | <void> | no | no | $fptag |
| FPIOFF | Instruction offset | <void> | no | no | $fpioff |
| FPISEL | Instruction selector | <void> | no | no | $fpisel |
| FPDOFF | Data offset | <void> | no | no | $fpdoff |
| FPDSEL | Data selector | <void> | no | no | $fpdsel |

## Intel-x86 FPCR Register

For your convenience, TotalView interprets the bit settings of the FPCR and FPSR registers.

You can edit the value of the FPCR and set it to any of the bit settings outlined in the next table.

Table 68: Intel-x86 FPCR Register Bit Settings

| Value | Bit Setting | Meaning |
|---|---|---|
| RC=NEAR | 0x0000 | To nearest rounding mode |
| RC=NINF | 0x0400 | Toward negative infinity rounding mode |
| RC=PINF | 0x0800 | Toward positive infinity rounding mode |
| RC=ZERO | 0x0c00 | Toward zero rounding mode |
| PC=SGL | 0x0000 | Single precision rounding |
| PC=DBL | 0x0080 | Double precision rounding |
| PC=EXT | 0x00c0 | Extended precision rounding |
| EM=PM | 0x0020 | Precision exception enable |
| EM=UM | 0x0010 | Underflow exception enable |
| EM=OM | 0x0008 | Overflow exception enable |
| EM=ZM | 0x0004 | Zero divide exception enable |
| EM=DM | 0x0002 | Denormalized operand exception enable |
| EM=IM | 0x0001 | Invalid operation exception enable |

## Using the Intel-x86 FPCR Register

You can change the value of the FPCR within TotalView to customize the exception handling for your program.

For example, if your program inadvertently divides by zero, you can edit the bit setting of the FPCR register in the Stack Frame Pane. In this case, you would change the bit setting for the FPCR to include 0x0004 (as shown in Table 68) so that TotalView traps the "divide by zero" bit. The string displayed next to the FPCR register should now include EM=(ZM). Now, when your program divides by zero, it receives a SIGFPE signal, which you can catch with TotalView. See Chapter 3 of the TOTALVIEW USER'S GUIDE for information on handling signals. If you did not set the bit for trapping divide by zero, the processor would ignore the error and set the EF=(ZE) bit in the FPSR.

## Intel-x86 FPSR Register

The bit settings of the Intel-x86 FPSR register are outlined in the following table.

Table 69: Intel-x86 FPSR Register Bit Settings

| Value | Bit Setting | Meaning |
|---|---|---|
| TOP=<*i*> | 0x3800 | Register <*i*> is top of FPU stack |
| B | 0x8000 | FPU busy |
| C0 | 0x0100 | Condition bit 0 |
| C1 | 0x0200 | Condition bit 1 |
| C2 | 0x0400 | Condition bit 2 |
| C3 | 0x4000 | Condition bit 3 |
| ES | 0x0080 | Exception summary status |
| SF | 0x0040 | Stack fault |
| EF=PE | 0x0020 | Precision exception |
| EF=UE | 0x0010 | Underflow exception |
| EF=OE | 0x0008 | Overflow exception |
| EF=ZE | 0x0004 | Zero divide exception |
| EF=DE | 0x0002 | Denormalized operand exception |
| EF=IE | 0x0001 | Invalid operation exception |

## Intel-x86 Floating-Point Format

The Intel-x86 processor supports the IEEE floating point format.

# Glossary

**ACTION POINT:** A debugger feature that allows a user to request that program execution stop under certain conditions. Action points include breakpoints, watchpoints, evaluation points, and barriers.

**ACTION POINT IDENTIFIER:** A unique integer ID associated with an action point.

**ADDRESS SPACE:** A region of memory that contains code and data from a program. One or more threads can run in an address space. A process normally contains an address space.

**AFFECTED P/T SET:** The set of threads that will be affected by the command. For most commands, this is identical to the target p/t set, but in some cases it may include additional threads.

**AGGREGATED OUTPUT:** The CLI compresses output from multiple threads when they would be identical except for the p/t identifier.

**ARENA:** A specifier that indicates the processes, threads, and groups upon which a command executes. Arena specifiers are **p** (process), **t** (thread), **g** (group), **d** (default), and **a** (all).

**AUTOMATIC PROCESS ACQUISITION:** TotalView automatically detects the many processes that parallel and distributed programs run in, and attaches to them automatically so you do not have to attach to them manually. This process is called *automatic process acquisition*. If the process is on a remote machine, automatic process acquisition automatically starts the TotalView debugger server (the tvdsvr).

**BARRIER:** An action point specifying that processes reaching a particular location in the source code should stop and wait for other processes to catch up.

**BREAKPOINT:** A point in a program where execution can be suspended to permit examination and manipulation of data.

**CALL STACK:** A higher-level view of stack memory, interpreted in terms of source program variables and locations.

**CHILD PROCESS:** A process created by another process (*see* **parent process**) when that other process calls **fork()**.

**CLUSTER DEBUGGING:** The action of debugging a program that is running on a cluster of hosts in a network. Typically, the hosts are homogeneous.

**COMMAND HISTORY LIST:** A debugger-maintained list storing copies of the most recent commands issued by the user.

**CONTEXTUALLY QUALIFIED (SYMBOL):** A symbol that is described in terms of its dynamic context, rather than its static scope. This includes process identifier, thread identifier, frame number, and variable or subprocedure name.

**CORE FILE:** A file containing the contents of memory and a list of thread registers. The operating system dumps (creates) a core file whenever a program exits because of a severe error (such as an attempt to store into an invalid address).

**CORE-FILE DEBUGGING:** A debugging session that examines a core file image. Commands that modify program state are not permitted in this mode.

**CROSS-DEBUGGING:** A special case of remote debugging where the host platform and the target platform are different types of machines.

**CURRENT FRAME:** The current portion of stack memory, in the sense that it contains information about the subprocedure invocation that is currently executing.

**CURRENT LANGUAGE:** The source code language used by the file containing the current source location.

**CURRENT LIST LOCATION:** The location governing what source code will be displayed in response to a list command.

**DATA-SET:** A set of array elements generated by TotalView and sent to the Visualizer. (See **visualizer process**.)

**DBELOG LIBRARY:** A library of routines for creating event points and generating event logs from within TotalView. To use event points, you must link your program with both the **dbelog** and **elog** libraries.

**DBFORK LIBRARY:** A library of special versions of the **fork()** and **execve()** calls used by the TotalView debugger to debug multiprocess programs. If you link your program with TotalView's **dbfork** library, TotalView will be able to automatically attach to newly spawned processes.

**DEBUGGING INFORMATION:** Information relating an executable to the source code from which it was generated.

**DEBUGGER INITIALIZATION FILE:** An optional file establishing initial settings for debugger state variables, user-defined commands, and any commands that should be executed whenever TotalView or the CLI is invoked. Must be called **.tvdrc**.

**DEBUGGER PROMPT:** A string printed by the CLI that indicates that it is ready to receive another user command.

**DEBUGGER SERVER:** *See* **tvdsvr process**.

**DEBUGGER STATE:** Information that TotalView or the CLI maintains in order to interpret and respond to user commands. Includes debugger modes, user-defined commands, and debugger variables.

**DISTRIBUTED DEBUGGING:** The action of debugging a program that is running on more than one host in a network. The hosts can be homogeneous or heterogeneous. For example, programs written with message-passing libraries such as Parallel Virtual Machine (PVM) or Parallel Macros (PARMACS) run on more than one host.

**DIVE STACK:** A series of nested dives that were performed in the same variable window. The number of greater than symbols (>) in the upper left-hand corner of a variable window indicates the number of nested dives on the dive stack. Each time that you undive, TotalView pops a dive from the dive stack and decrements the number of greater than symbols shown in the variable window.

**DIVING:** The action of displaying more information about an item. For example, if you dive into a variable in TotalView, a window appears with more information about the variable.

**EDITING CURSOR:** A black rectangle that appears when a TotalView GUI field is selected for editing. You use field editor commands to move the editing cursor.

**EVALUATION POINT:** A point in the program where TotalView evaluates a code fragment without stopping the execution of the program.

**EVENT LOG:** A file containing a record of events for each process in a program.

**EVENT POINT:** A point in the program where TotalView writes an event to the event log for later analysis using TimeScan.

**EXECUTABLE:** A compiled and linked version of source files, containing a "main" entry point.

**EXPRESSION:** An expression consists of symbols (possibly qualified), constants, and operators, arranged in the syntax of the current source language. Not all Fortran 90, C, and C++ operators are supported.

**EXTENT:** The number of elements in the dimension of an array. For example, a Fortran array of integer(7,8) has an extent of 7 in one dimension (7 rows) and an extent of 8 in the other dimension (8 columns).

**FIELD EDITOR:** A basic text editor that is part of TotalView's interface. The field editor supports a subset of GNU Emacs commands.

**FOCUS:** The set of groups, processes, and threads upon which a CLI command acts. The current focus is indicated in the CLI prompt (if you are using the default prompt).

**FRAME:** An area in stack memory containing the information corresponding to a single invocation of a subprocedure.

**FULLY QUALIFIED (SYMBOL):** A symbol is fully qualified when each level of source code organization is included. For variables, those levels are executable or library, file, procedure or line number, and variable name.

**GRIDGET:** A dotted grid in the tag field that indicates you can set an action point on the instruction.

**GROUP:** When TotalView starts processes, it places related processes in families. These families are called "groups."

**HOST MACHINE:** The machine on which the TotalView debugger is running.

**INITIAL PROCESS:** The process created as part of a load operation, or that already existed in the run-time environment and was attached by TotalView or the CLI.

**LVALUE:** A symbol name or expression suitable for use on the left-hand side of an assignment statement in the corresponding source language. That is, the expression must be appropriate as the target of an assignment.

**LHS EXPRESSION:** This is a synonym for **lvalue**.

**LOWER BOUND:** The first element in the dimension of an array or the slice of an array. By default, the lower bound of an array is 0 in C and 1 in Fortran, but the lower bound can be any number, including negative numbers.

**MACHINE STATE:** Convention for describing the changes in memory, registers, and other machine elements as execution proceeds.

**MESSAGE QUEUE:** A list of messages sent and received by message-passing programs.

**MPICH:** MPI/Chameleon (Message Passing Interface/Chameleon) is a freely available and portable MPI implementation. MPICH was written as a collaboration between Argonne National Lab and Mississippi State University. For more information, see **www.mcs.anl.gov/mpi**.

**MPMD (MULTIPLE PROGRAM MULTIPLE DATA) PROGRAMS:** A program involving multiple executables, executed by multiple threads and processes.

**MUTEX:** Mutual exclusion. A collection of techniques for sharing resources so that different uses do not conflict and cause unwanted interactions.

**NATIVE DEBUGGING:** The action of debugging a program that is running on the same machine as TotalView.

**NESTED DIVE WINDOW:** A TotalView window that results from diving into an item in a variable window. A nested dive window replaces the contents of the variable window and has an undive symbol in its title bar. Diving on the undive symbol returns the original contents of the variable window.

**OUT OF SCOPE:** When symbol lookup is performed for a particular symbol name and it is not found in the current scope or any containing scopes, the symbol is said to be out of scope.

**PARALLEL PROGRAM:** A program whose execution involves multiple threads and processes.

**PARCEL:** The number of bytes required to hold the shortest instruction for the target architecture.

**PARENT PROCESS:** A process that calls **fork()** to spawn other processes (usually called child processes).

**PARMACS LIBRARY:** A message-passing library for creating distributed programs that was developed by the German National Research Centre for Computer Science.

**PARTIALLY QUALIFIED (SYMBOL):** A symbol name that includes only some of the levels of source code organization (for example, filename and procedure, but not executable). This is permitted as long as the resulting name can be associated unambiguously with a single entity.

**PC:** This is an abbreviation for P*rogram* C*ounter*.

**PROCESS:** An executable that is loaded into memory and is running (or capable of running).

**PROCESS GROUP:** A group of processes associated with a multiprocess program. A process group includes program groups and share groups.

**PROCESS/THREAD IDENTIFIER:** A unique integer ID associated with a particular process and thread.

**PROGRAM EVENT:** A program occurrence that is being monitored by TotalView or the CLI, such as a breakpoint.

**PROGRAM GROUP:** A group of processes that includes the parent process and all related processes. A program group includes children that were forked (processes that share the same source code as the parent) and children that were forked with a subsequent call to **execve()** (processes that do *not* share the same source code as the parent). Contrast with **share group**.

**PROGRAM STATE:** A higher-level view of the machine state, where addresses, instructions, registers, and such are interpreted in terms of source program variables and statements.

**P/T (PROCESS/THREAD) SET:** The set of threads drawn from all threads in all processes of the target program.

**PVM LIBRARY:** Parallel Virtual Machine library. A message-passing library for creating distributed programs that was developed by the Oak Ridge National Laboratory and the University of Tennessee.

**RVALUE:** An expression suitable for inclusion on the right-hand side of an assignment statement in the corresponding source language. In other words, an expression that evaluates to a value or collection of values.

**REMOTE DEBUGGING:** The action of debugging a program that is running on a different machine than TotalView. The machine on which the program is running can be located many miles away from the machine on which TotalView is running.

**RESUME COMMANDS:** Commands that cause execution to restart from a stopped state: **dstep**, **dgo**, **dcont**, **dwait**.

**RHS EXPRESSION:** This is a synonym for **rvalue**.

**RUNNING STATE:** The state of a thread when it is executing, or at least when the CLI or TotalView has passed a request to the underlying run-time system that the thread be allowed to execute.

**SERIAL LINE DEBUGGING:** A form of remote debugging where TotalView and the TotalView Debugger Server communicate over a serial line.

**SHARE GROUP:** A group of processes that includes the parent process and any related processes that share the same source code as the parent. Contrast with **program group**.

**SHARED LIBRARY:** A compiled and linked set of source files that are dynamically loaded by other executables—and have no "main" entry point.

**SIGNALS:** Messages informing processes of asynchronous events, such as serious errors. The action the process takes in response to the signal depends on the type of signal and whether or not the program includes a signal handler routine, a routine that traps certain signals and determines appropriate actions to be taken by the program.

**SINGLE STEP:** The action of executing a single statement and stopping (as if at a breakpoint).

**SLICE:** A subsection of an array, which is expressed in terms of a lower bound, upper bound, and stride. Displaying a slice of an array can be useful when working with very large arrays, which is often the case in Fortran programs.

**SOURCE FILE:** Program file containing source language statements. TotalView allows you to debug FORTRAN 77, Fortran 90, Fortran 95, C, C++, and assembler.

**SOURCE LOCATION:** For each thread, the source code line it will execute next. This is a static location, indicating the file and line number; it does not, however, indicate which invocation of the subprocedure is involved.

**SPAWNED PROCESS:** The process created by a user process executing under debugger control.

**SPMD (SINGLE PROGRAM MULTIPLE DATA) PROGRAMS:** A program involving just one executable, executed by multiple threads and processes.

**STACK:** A portion of computer memory and registers used to hold information temporarily. The stack consists of a linked list of stack frames that holds return locations for called routines, routine arguments, local variables, and saved registers.

**STACK FRAME:** A section of the stack that contains the local variables, arguments, contents of the registers used by an individual routine, a frame pointer pointing to the previous stack frame, and the value of the Program Counter (PC) at the time the routine was called.

**STACK POINTER:** A pointer to the area of memory where subprocedure arguments, return addresses, and similar information is stored.

**STACK TRACE:** A sequential list of each currently active routine called by a program and the frame pointer pointing to its stack frame.

**STATIC (SYMBOL) SCOPE:** A region of a program's source code that has a set of symbols associated with it. A scope can be nested inside another scope.

**STEPPING:** Advancing program execution by fixed increments, such as by source code statements.

**STOP SET:** A set of threads that should be stopped once an action point has been triggered.

**STOPPED/HELD STATE:** The state of a process whose execution has paused in such a way that another program event (for example, arrival of other threads at the same barrier) will be required before it is capable of continuing execution.

**STOPPED/RUNNABLE STATE:** The state of a process whose execution has been paused (for example, when a breakpoint triggered or due to some user command) but can continue executing as soon as a resume command is issued.

**STOPPED STATE:** The state of a process that is no longer executing, but will eventually execute again. This is subdivided into stopped/runnable and stopped/held.

**STRIDE:** The interval between array elements in a slice and the order in which the elements are displayed. If the stride is 1, every element between the lower bound and upper bound of the slice is displayed. If the stride is 2, every other element is displayed. If the stride is –1, every element between the upper bound and lower bound (reverse order) is displayed.

**SYMBOL:** Entities within program state, machine state, or debugger state.

**SYMBOL LOOKUP:** Process whereby TotalView consults its debugging information to discover what entity a symbol name refers to. Search starts with a particular static scope and occurs recursively, so that containing scopes are searched in an outward progression.

**SYMBOL NAME:** The name associated with a symbol known to TotalView (for example, function, variable, data type, and such).

**SYMBOL TABLE:** A table of symbolic names (such as variables or functions) used in a program and their memory locations. The symbol table is part of the executable object generated by the compiler (with the **–g** switch) and is used by debuggers to analyze the program.

**TAG FIELD:** The left margin in the source code pane of the TotalView process window containing boxed line numbers marking the lines of source code that actually generate executable code.

**TARGET MACHINE:** The machine on which the process to be debugged is running.

**TARGET PROCESS SET:** The target set for those occasions when operations may only be applied to entire processes, not to individual threads within a process.

**TARGET PROGRAM:** The executing program that is the target of debugger operations.

**TARGET P/T SET:** The set of processes and threads upon which a CLI command will act.

**THREAD:** An execution context that normally contains a set of private registers and a region of memory reserved for an execution stack. A thread runs in an address space.

**THREAD EXECUTION STATE:** The convention of describing the operations available for a thread, and the effects of the operation, in terms of a set of pre-defined states.

**THREAD OF INTEREST:** The primary thread that will be affected by a command.

**TRIGGER SET:** The set of threads that may trigger an action point (that is, for which the action point was defined).

**TRIGGERS:** The effect during execution when program operations cause an event to occur (such as, arriving at a breakpoint).

**TVDSVR PROCESS:** The TotalView Debugger Server process, which facilitates remote debugging by running on the same machine as the executable and communicating with TotalView over a TCP/IP port or serial line.

**UNDIVING:** The action of displaying the previous contents of a window, instead of the contents displayed for the current dive. To undive, you dive on the undive icon in the upper right-hand corner of the window.

**UPPER BOUND:** The last element in the dimension of an array or the slice of an array.

**USER INTERRUPT KEY:** A keystroke used to interrupt commands, most commonly defined as ^C (Ctrl-C).

**VARIABLE WINDOW:** A TotalView window displaying the name, address, data type, and value of a particular variable.

**VISUALIZER PROCESS:** A process that works with TotalView in a separate window, allowing you to see a graphical representation of program array data.

**WATCHPOINT:** An action point specifying that execution should stop whenever the value of a particular variable is updated.

# Index

## Symbols

## B

## E

## F

## G



## H

## J



## K



## L

# N

## Q



## R

## S

## T

## U



## V

## W



## X



## Z